National Library of Canada
Canadian Theses Service
Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Service des thèses canadiennes

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

The University of Alberta

SSE: An Integrated High Level Synthesis System

by

Tai A. Ly ©

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Doctor of Philosophy

Department of Electrical Engineering

Edmonton, Alberta
Spring, 1992

National Library
of Canada

Canadian Theses Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale
du Canada

Service des thèses canadiennes

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-73024-2

Canadä

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Tai A. Ly

TITLE OF THESIS: SSE: An Integrated High Level Synthesis System

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Doctor of Philosophy

YEAR THIS DEGREE GRANTED: 1992

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) ....
Permanent Address:
2023 - 52 Street
Edmonton, Alberta
Canada T6L 2G9

Dated 6 December 1991

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **SSE: An Integrated High Level Synthesis System** submitted by **Tai A. Ly** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**

Jack Mowchenko

Dr. J. T. Mowchenko

Prof. W. B. Joerg

J Schaeffer

Dr. J. Schaeffer

Prof P.W. Lawson

PP Dr. M. McFarland

T.A. Marsland

Dr. T. A. Marsland

Date Nov. 30, 1991

ABSTRACT

Traditional high level synthesis systems divide the synthesis problem into a number of simpler tasks, which are then solved separately in a fixed order. While this reduces the complexity of the synthesis algorithms, it does so at the expense of synthesis quality because the synthesis tasks are heavily interdependent on one another. In this thesis, we propose a novel framework for high level synthesis in which different synthesis tasks may be solved concurrently. To demonstrate the feasibility of such a system, we integrate the synthesis tasks of scheduling and allocation in the prototype **SSE** (for Synthesis by Simulated Evolution) system. Other synthesis tasks can then be similarly integrated to this system in the future.

The SSE synthesis system features:

1. New Simulated Evolution-based scheduling and allocation algorithms which, despite their local heuristics and simple cost functions, produce good designs very quickly by effectively exploring the design space.

2. A generalized formulation of the allocation task which considers unscheduled operations as having *uncertain* schedules and applies fuzzy set theory to represent and reason about such uncertainties, thereby extending the allocation algorithm to handle incomplete schedules.

3. An integration of the above scheduling and allocation algorithms as concurrent, algorithmic *agents* in a Blackboard architecture, allowing more sophisticated synthesis strategies than are possible in previous synthesis systems.

Acknowledgements

I would like to thank Dr. Emil Girczyc for opening my eyes to the field of high level synthesis, and for his encouragement in the early stages of this research. I also thank my wife Christina for her love and support throughout my graduate school years.

Table of Content

Chapter 1. Introduction 1
  1.1. High Level Synthesis 1
    1.1.1. Synthesis Input 2
    1.1.2. Synthesis Output 3
    1.1.3. Synthesis Objectives 4
    1.1.4. Synthesis Subtasks 4
  1.2. Thesis Goals 7
  1.3. Organization of Thesis 8
Chapter 2. System Overview 11
  2.1. Traditional Synthesis Process 11
  2.2. Proposed Synthesis System 13
  2.3. Prototype System Overview 15
    2.3.1. Applying Simulated Evolution to Scheduling and Allocation 16
    2.3.2. Allocation based on Fuzzy Schedules 16
    2.3.3. Integrating Scheduling and Allocation 17
    2.3.4. Assumptions 18
    2.3.5. Design Representations 19
      2.3.5.1. CDFG 20
      2.3.5.2. CG 24
Chapter 3. Related Work 26
  3.1. Separate Scheduling and Allocation Systems 26
  3.2. Concurrent Scheduling and Allocation Systems 29
  3.3. Transformational Synthesis Systems 30
  3.4. Comparison 31
Chapter 4. Optimization by Simulated Evolution 35
  4.1. The SE Algorithm 35
  4.2. Optimum Assignment Problem 38
  4.3. Implementation of GENERATE 39
  4.4. Implementation of SELECT 40
  4.5. Summary 42
Chapter 5. Basic Scheduling Algorithm 43
  5.1. Previous Scheduling Approaches 43
  5.2. Scheduling as an Optimum Assignment Problem 46
  5.3. The PRIORITY Function 49
  5.4. The INCR Function 51
    5.4.1. Incremental Operator Cost 51
    5.4.2. Incremental Register Cost 52
    5.4.3. Incremental Interconnect Cost 54
    5.4.4. Incremental Opportunity Cost 55

5.5. The GLOBAL Function 56
5.6. Summary 58
Chapter 6. Basic Allocation Algorithm 59
6.1. Previous Allocation Approaches 59
6.2. Allocation as an Optimum Assignment Problem 60
6.3. The PRIORITY Function 63
6.4. The INCR Function 65
6.5. The GLOBAL Function 67
6.6. Summary 68
Chapter 7. Refined Synthesis Algorithms 70
7.1. Hardware Constraints 71
7.2. Local Timing Constraints 71
7.3. Case Constructs 72
7.4. Loops 72
7.5. Area/Time Tradeoffs 72
7.6. Operation Chaining 73
7.7. Equivalent Allocation Constraints 74
7.8. Operators with Permutable Inputs 75
7.9. Interconnect Optimization 75
7.10. Catastrophic Rip-Up in Allocation 76
7.11. Structural Pipelining 76
7.12. Algorithmic Pipelining 77
Chapter 8. Design Examples for SE-based Synthesis 79
8.1. Differential Equation Example from HAL 79
8.2. Pipelined Examples from Sehwa 84
8.3. Fifth Order Elliptic Wave Filter (EWF) Example 86
8.4. Summary 94
Chapter 9. Comparing SE to Simulated Annealing 95
9.1. Simulated Annealing 95
9.2. An Intuitive Comparison of SE and SA 97
9.3. Experiments on Effects of Guided Search 98
9.4. Experiments on Effects of State Transition Distance 102
9.5. Combining the SE and SA Algorithms 106
9.6. Summary 108
Chapter 10. Bottom Up Synthesis based on Fuzzy Schedules 110
10.1. Introduction 110
10.2. Fuzzy Schedulability 113
10.2.1. Fuzzy Set Theory 113
10.2.2. Fuzzy Schedule 114
10.2.3. Fuzzy Lifetimes 118
10.2.4. Fuzzy Schedulability 121

10.2.5. Allocation using Fuzzy Schedulability 123
10.3. Implementation Issues 125
10.3.1. Statistical Fuzzy Schedules 125
10.3.2. Approximate Schedulability Decrement 127
10.3.3. Dependency Analysis 127
10.3.4. Interval Analysis 128
10.4. Experimental Results 130
10.5. Discussions 134
10.5.1. Comparing Fuzzy Allocation with Design Partitioning 134
10.5.2. Optimistic and Pessimistic Formulations 135
10.5.3. Scheduling with Partial Allocation 135
10.6. Summary 137
Chapter 11. Integrating Scheduling and Allocation 139
11.1. Blackboard and Kernel Software 140
11.2. Algorithmic Agents 141
11.3. Control Mechanisms 142
11.3.1. Intermixing of Synthesis Steps 142
11.3.2. Focusing of Attention 144
11.4. Experiments with Synthesis Strategies 146
11.4.1. Fine-Grained Integration 146
11.4.2. Top-Down and Bottom-Up Synthesis with Iterations 147
11.4.3. Allocation based on Iterative Improvement 147
11.4.4. Goal Directed Synthesis 150
11.5. Summary 154
Chapter 12. Conclusion 155
12.1. Summary 155
12.2. Major Contributions 157
12.3. Suggestions for Future Work 158
References 160

List of Tables

Table 8.1 Design Costs for Differential Equation Example (4 Control Steps) 82
Table 8.2 Design Costs for Differential Equation Example (8 Control Steps) 82
Table 8.3 Schedule Costs for the FIR Filter Example 86
Table 8.4 Scheduling Performance for the EWF Example 89
Table 8.5 Design Costs for the EWF Example 90
Table 8.6 Schedule and Allocation for the EWF Example 91
Table 8.7 Scheduling Performance for the EWF Unrolled 3 Times 94
Table 10.1 Experimental Results on the EWF Example (19 Control Steps) 133
Table 11.1 Best Schedule and Allocation for the EWF Example 152

List of Figures

Figure 1.1 Design Synthesis Models in the Y-Diagram 2
Figure 1.2 Sample HDL Code Segment 3
Figure 1.3 A Simple Synthesis Example 6
Figure 2.1 Traditional Synthesis Process 12
Figure 2.2 Complete Proposed System for Integrated High Level Synthesis 14
Figure 2.3 Two-Phase Clocking Scheme 19
Figure 2.4 Sample CDFG Data Nodes and Data Edges 20
Figure 2.5 Sample HDL and CDFG with a LOOP Construct 21
Figure 2.6 Sample HDL and CDFG with a CASE Construct 22
Figure 2.7 Example of (a) Minimum and (b) Maximum Timing Constraints 23
Figure 2.8 A Sample CG with 2 Functional Units, 3 Registers, a Mux and a Bus 24
Figure 4.1 Pseudo-Code for the SE Algorithm 36
Figure 4.2 Pseudo-Code for GENERATE 39
Figure 4.3 Pseudo-Code for SELECT 41
Figure 7.1 Algorithmic Pipelining: Consecutive Instances of a LOOP Overlap in Time 77
Figure 8.1 CDFG for the Differential Equation Example 80
Figure 8.2 Circuits for Differential Equation Example (4 Control Steps) 81
Figure 8.3 Circuits for Differential Equation Example (8 Control Steps) 83
Figure 8.4 Schedules for the FIR Filter Example 85
Figure 8.5 Scheduled Example with Algorithmic Pipeline and CASE Constructs 87
Figure 8.6 CDFG for the EWF Example 88
Figure 8.7 Circuits for the EWF Example 92
Figure 8.8 Run Time Profile for SE-based Allocation 93
Figure 9.1 Pseudo-Code for the SA Algorithm 96
Figure 9.2 Performance with Randomized PRIORITY, INCR and GLOBAL 99
Figure 9.3 Performance Without Component Costs in INCR (Case 1) 100
Figure 9.4 Performance Without Component Costs in INCR (Case 2) 101
Figure 9.5 Pseudo-Code for SA-GENERATE 102
Figure 9.6 Statistical Performance for SA-Based Scheduling 103
Figure 9.7 SA Failures vs. Maximum State Transition Distance 104
Figure 9.8 SE Failures vs. Maximum State Transition Distance 105
Figure 9.9 Pseudo-Code for the Combined SE/SA Algorithm 107
Figure 9.10 Performance of SE/SA Based Scheduling for Select T_0 Values 107
Figure 10.1 CDFG Segment from the EWF Example 120

Figure 10.2 $S_n(i)$ for n = +27, +29 and x30 120
Figure 10.3 $l_n(i)$ for n = +27, +29 and x30 121
Figure 10.4 $eb_n(i+1)$ for n = +27, +29 and x30 121
Figure 10.5 $sa_n(i-d(n)+1)$ for n = +27, +29 and x30 122
Figure 10.6 $l_x(i)$ for x = e27 and e29 122
Figure 10.7 Sample Statistical Performance for the EWF 131
Figure 11.1 The Blackboard/Agent Architecture in SSE 139
Figure 11.2 Pseudo-Code for Iterative Improvement Based Allocation 148
Figure 11.3 Allocation Performance for Select Values of m and n 149
Figure 11.4 Best Circuit for the EWF Example 153

Chapter 1. Introduction

As VLSI technology advances and IC designs grow more and more complex, the importance of *design synthesis* tools becomes increasingly apparent. In the most general sense, *design synthesis* refers to the automatic generation of a design description from another design description at a higher level of abstraction. At the lower levels, *physical design* tools generate final layout of a circuit from a structural *netlist* of a design (i.e., interconnected cell structures). At a higher level of abstraction, *logic synthesis* tools generate a structural netlist from a logic equation description of a design. At the next higher level is the *register transfer level* (RTL) of design descriptions, which consist of interconnected functional units (e.g., adders, ALUs, multipliers), registers, multiplexers, busses, and a finite state machine (FSM) description of the timing and control of these interconnected cells. *RTL synthesis* tools generate logic equations and structural netlists from RTL design descriptions. Finally, above RTL are the *behavioral* and *algorithmic* levels of design descriptions. Synthesis from these levels of design abstraction is referred to as *high level synthesis*, which is the subject of this thesis.

1.1. High Level Synthesis

High level synthesis is concerned with the automatic generation of RTL design descriptions from the *algorithmic* or behavioral specifications of digital systems. This is called high level synthesis because it is at a higher level of design abstraction than RTL synthesis, logic synthesis and physical design. Fig. 1.1 depicts these design processes on the "Y-diagram" of design representations [47,58].

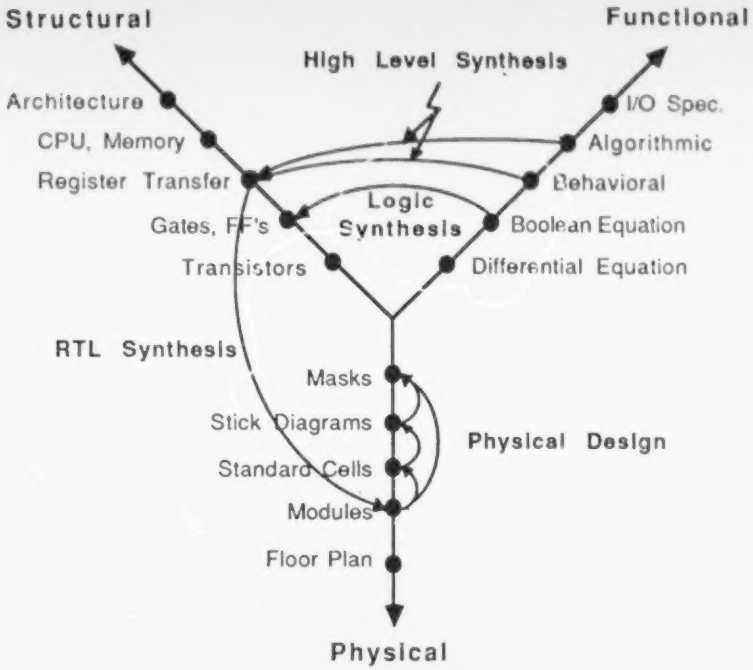

Structural
Functional
High Level Synthesis
Architecture
CPU, Memory
Register Transfer
Gates, FF's
Transistors
I/O Spec.
Algorithmic
Behavioral
Boolean Equation
Differential Equation
Logic Synthesis
RTL Synthesis
Masks
Stick Diagrams
Standard Cells
Modules
Floor Plan
Physical Design
Physical

Figure 1.1 Design Synthesis Models in the Y-Diagram

1.1.1. Synthesis Input

An *algorithmic* description of a system specifies the input/output mapping of the system in terms of functional operations and possibly timing constraints on these operations. A *behavioral* description, on the other hand, specifies not only the functionality and timing of a system, but also certain structures and/or internal behaviors of the system [47]. Fig. 1.2 shows a hardware description language (HDL) code segment for a system that computes the sum of its four inputs. This code segment is an algorithmic specification if we ignore the sequencing of code statements, and ignore the fact that both statements 2 and 4 assign values to variable $t1$. However, this code segment is a behavioral specification if we observe source code sequencing (i.e., operation $A+B$ is performed before operation $C+D$), or interpret multiple assignments to the same variable name (i.e., $t1$) as storing values to the same memory element (i.e., statements 2 and 4 both write to the same memory element).

```
1:  READ(A);  READ(B);  READ(C);  READ(D);
2:  t1  <-  A  +  B;
3:  t2 <-  C  +  D;
4:  t1  <-  t1  +  t2;
```

Figure 1.2 Sample HDL Code Segment

1.1.2. Synthesis Output

The output of high level synthesis is an RTL description which typically specifies a data path and a control path. The data path is a circuit containing all functional units (e.g., adders, multipliers and comparators), memory elements (e.g., registers, register files and RAM) and interconnects (e.g., busses, multiplexers and signal nets) required to perform all data transfer, data storage and functional operations in the input specification. The control path is a finite state machine (FSM) which produces all control signals required for the data path cells. Typically, control paths are implemented by automatic FSM generators based on microprogram or PLA based controller architectures. This RTL circuit description is then passed to lower level design tools for final layout.

1.1.3. Synthesis Objectives

The objectives of high level synthesis, like that of any other design activity, are to minimize costs subject to performance constraints, or maximize performance subject to cost constraints, or a combination of both. In this context, costs of a circuit may include hardware area, testability, power consumption, etc, whereas the performance of a design may be measured in terms of throughput and/or response time. For the purpose of this thesis, we measure the cost of a design by the total active area (e.g., measured by the number of gates used) in its data path, and we measure the performance of a design by its throughput. Consequently, given an algorithmic/behavioral specification, we either minimize the data path area subject to a maximum constraint on the time delay between inputs and outputs, or minimize this time delay subject to a maximum constraint on the data path area required.

1.1.4. Synthesis Subtasks

We identify six subtasks in high level synthesis:

1. **Translation:** This translates the input specification from a high-level HDL to a *Control and Data Flow Graph* (CDFG) representation. This makes explicit all data and control dependencies among operations and variables in the input specifications, and removes any ambiguity that may arise from source code sequencing and multiple assignments to named variables. Fig. 1.3(a) shows a sample HDL code segment, and Fig. 1.3(b) shows the equivalent CDFG representation for this HDL code. By itself, the CDFG is an algorithmic specification of the design. However, if we permit users to specify explicit bindings of operations (or data values) to clock cycles and/or hardware cells, and denote these bindings in the CDFG, then the output of this translation step may be a behavioral specification of the design.

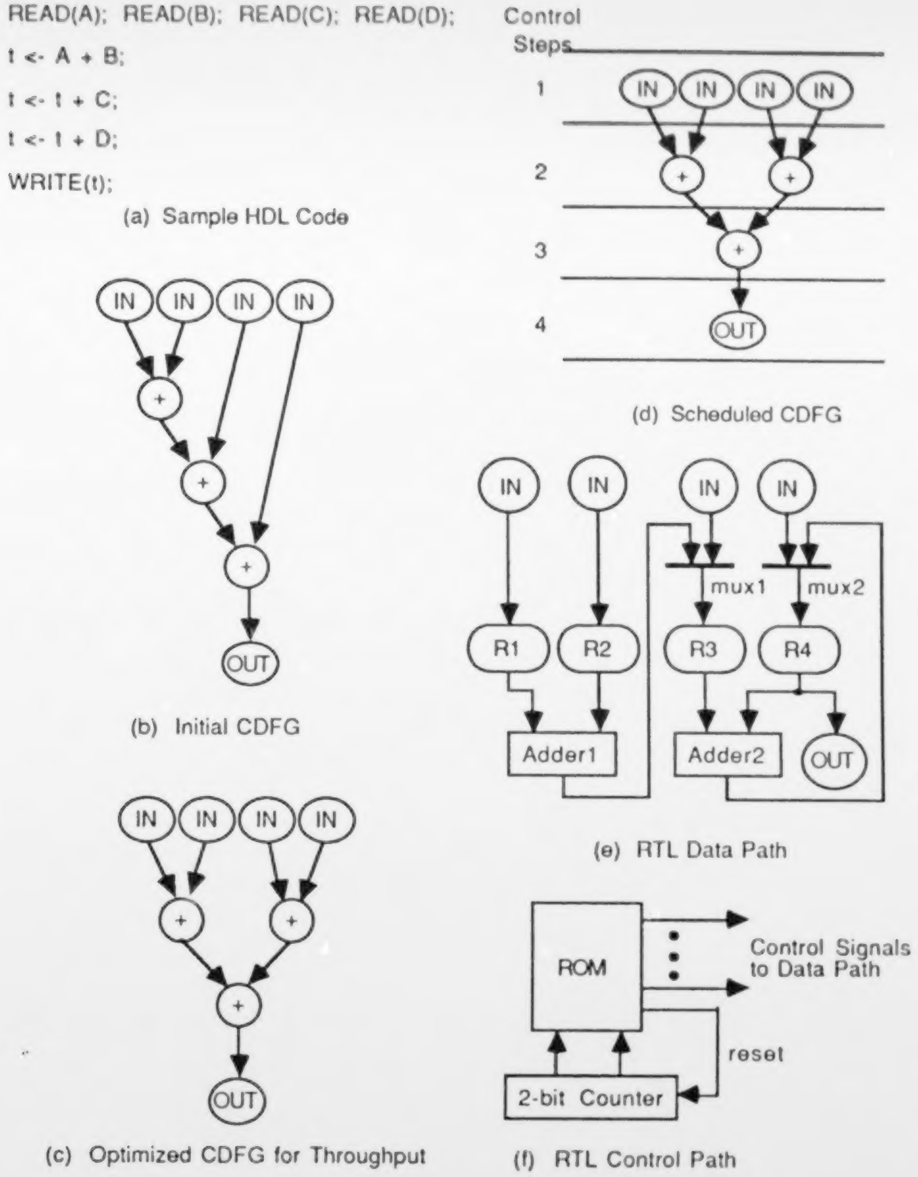
2. **Functional Optimization:** This applies correctness preserving transformations to the CDFG representation of the input specification, or simply CDFG, to optimize the CDFG for subsequent synthesis steps. Fig. 1.3(c) shows a CDFG which is optimized with respect to circuit throughput from the CDFG of Fig. 1.3(b).

3. **Module Selection:** This selects a set of library cells, or modules, to implement the operations, store the data values, and realize the data transfers in the (optimized) CDFG.

4. **Scheduling:** This sequences all operations in the CDFG by assigning them to specific clock cycles, or *control steps*, such that all data and control dependencies are satisfied. Fig. 1.3(d) shows the scheduled CDFG of Fig. 1.3(c).

5. **Allocation:** This instantiates and assigns specific hardware cells to operations and data values in the CDFG, and makes all circuit interconnects to implement data transfers among these cells. Fig. 1.3(e) shows the data path created by allocation for the scheduled CDFG in Fig. 1.3(d).

6. **Control Path Generation** This determines the specification for the control path FSM by extracting all control signals required in each control step, and then passes this specification to an automatic FSM generator. Fig. 1.3(f) shows the resulting control path which, together with the data path of Fig. 1.3(e), makes up the final implementation for the input specification in Fig. 1.3(a).



```
READ(A); READ(B); READ(C); READ(D);
t <- A + B;
t <- t + C;
t <- t + D;
WRITE(t);
```

(a) Sample HDL Code

(b) Initial CDFG

(c) Optimized CDFG for Throughput

(d) Scheduled CDFG

(e) RTL Data Path

(f) RTL Control Path

IN
+
OUT
Control Steps
1
2
3
4
mux1
mux2
R1
R2
R3
R4
Adder1
Adder2
ROM
Control Signals to Data Path
reset
2-bit Counter

Figure 1.3 A Simple Synthesis Example

1.2. Thesis Goals

In this thesis, we will describe a novel framework in which different tasks in high level synthesis may be solved concurrently. To demonstrate the feasibility of such a system, we integrate the tasks of scheduling and allocation in a prototype implementation named **SSE** (for *Synthesis by Simulated Evolution*). Other synthesis tasks, such as functional optimization and module selection, may then be similarly integrated into this system in the future. Early work on different components of this system has been reported in [31, 32, 34-36].

This work has three major goals. The first goal is to show the value of *effective design space exploration* in scheduling and allocation. Previous work in scheduling and allocation has focused on developing more and more complex heuristics and cost functions to improve design quality at the expense of increased run times. However, we will show that simple heuristics and cost functions can produce good designs quickly if the CPU time is devoted to an effective exploration of the design space instead. In particular, we will apply the optimization technique of *simulated evolution* [22] to scheduling and allocation, and show that the resulting algorithms are simple, fast, and produce good designs by effective exploration of the design space.

The second goal of this work is to generalize the allocation task to allow for bottom-up synthesis, in which allocation is performed before scheduling. Previous work in high level synthesis has largely assumed a top-down approach, in which operations are only allocated after they have been scheduled, and data values are only allocated after their lifetimes are well defined (i.e., when the operation producing the data has been scheduled, and when the latest operation accessing the data has been scheduled). Only recently did work start on bottom-up synthesis [38, 54]. We will describe a new formulation of the allocation task which considers unscheduled operations as having *uncertain* schedules. Fuzzy set theory is then applied to such uncertainties, and the allocation task is generalized to that of finding the smallest circuit for which a successful schedule is still *highly possible*.

The third goal of this work is to demonstrate the feasibility of a framework for concurrent scheduling and allocation based on the above synthesis algorithms. We will describe how the simulated evolution-based scheduling and allocation algorithms may be organized as autonomous *agents* which work on the CDFG in a blackboard architecture. A supervisor module then dynamically intermixes scheduling and allocation steps at the level of individual operations and data values. We will also present a number of supervisor schemes to illustrate the power and flexibility of such a synthesis system.

1.3. Organization of Thesis

In "Chapter 2: System Overview", we will first describe the traditional top-down design process for high level synthesis, and propose a new synthesis system in which all synthesis subtasks are solved concurrently. We then give a brief overview of our prototype SSE system.

"Chapter 3: Related Work" gives a brief overview of other synthesis systems, classified according to the way they implement the interactions between the tasks of scheduling and allocation. We also give a summary comparison between these synthesis systems and the SSE system.

In "Chapter 4: Optimization by Simulated Evolution", we introduce the general optimization technique of Simulated Evolution (SE), and describe our implementation of SE in the context of an *optimum assignment problem*. This separates all application specific components of the SE algorithm into three cost functions, namely *PRIORITY*, *INCR* and *GLOBAL*.

This paves the way for "Chapter 5: Basic Scheduling Algorithm" and "Chapter 6: Basic Allocation Algorithm", which describe SE-based scheduling and allocation algorithms, respectively, by first formulating each synthesis task as an optimum assignment problem, and then defining the cost functions *PRIORITY*, *INCR* and *GLOBAL* for each task.

In "Chapter 7: Refined Synthesis Algorithms", we outline a number of extensions to the basic synthesis algorithms to accommodate "realistic" design constraints such as synthesis with pipelined functional units, hardware constraints, and local timing constraints. This shows that the SE-based synthesis algorithms can be readily extended to incorporate additional application features with straightforward changes to the cost functions.

"Chapter 8: Design Examples for SE-based Synthesis" presents experimental results produced by the SE-based scheduling and allocation algorithms on a number of design examples taken from the literature. This demonstrates that, compared to other synthesis systems, the SE-based synthesis algorithms generate comparable designs very quickly, and generate much better designs when given longer run-times.

In "Chapter 9: Comparing SE to Simulated Annealing", we compare SE and simulated annealing (SA) in the context of scheduling. We propose two hypotheses as to why SE may be better than SA, especially for highly constrained problems such as scheduling, and present experimental results in support of our hypotheses. In the process, we will describe a new implementation of SA which allows arbitrarily complex transformations modeled after the SE algorithm.

"Chapter 10: Bottom Up Synthesis based on Fuzzy Schedules" presents a new formulation of the allocation task which allows full allocation of all operations and data values prior to scheduling. We describe the implementation of this new formulation as a simple extension to our SE-based allocation algorithm. We then present experimental results which demonstrate the effectiveness of this new formulation for bottom-up synthesis.

In "Chapter 11: Integrating Scheduling and Allocation", we describe the integration of our SE-based scheduling and allocation in our SSE prototype system, and present a number of synthesis strategies which we have experimented with.

Finally, "Chapter 12: Conclusion" summarizes important contributions in this thesis, gives a few concluding remarks, and discusses possible directions for future research.

Chapter 2. System Overview

In this chapter, we first describe the traditional top-down design process for high level synthesis, and then propose a novel synthesis system in which all synthesis subtasks are solved by autonomous *agents* defined on the CDFG and organized in a blackboard architecture. This is followed by a brief overview of the prototype system, SSE, which is designed to demonstrate the feasibility of our proposed synthesis system.

2.1. Traditional Synthesis Process

Fig. 2.1 shows the top-down design process assumed by most high level synthesis systems. Given a HDL specification, the first step in high level synthesis is **translation** to a graph based representation such as CDFG, *value trace* [15], or ETPN [49]. This is followed by **functional optimization**, which applies optimization techniques such as inline expansion, dead-code elimination and code migration (into and out of *loops* and *case* constructs) [59] to the CDFG, in order to optimize the design at the algorithmic level without changing the function of the CDFG.

The next step is **module selection**, which selects a set of library cells which are to be used for implementing the final design [19]. This step usually selects one functional unit type for all operations of the same type, and one register type for all data values of the same bit width, although subsequent RTL synthesis tools may override such cell selection decisions by customizing individual cells.

The next two steps, **scheduling** and **allocation**, are the primary tasks in high level synthesis. The scheduling step sequences all operations by assigning these operations to start in specific *control steps*, such that all input data for each operation are available when the operation starts to execute. This defines the *lifetimes* of all operations and data values in the CDFG. (The lifetime of an operation is the sequence of control steps in which the operation will be executing, whereas the lifetime of a data value is

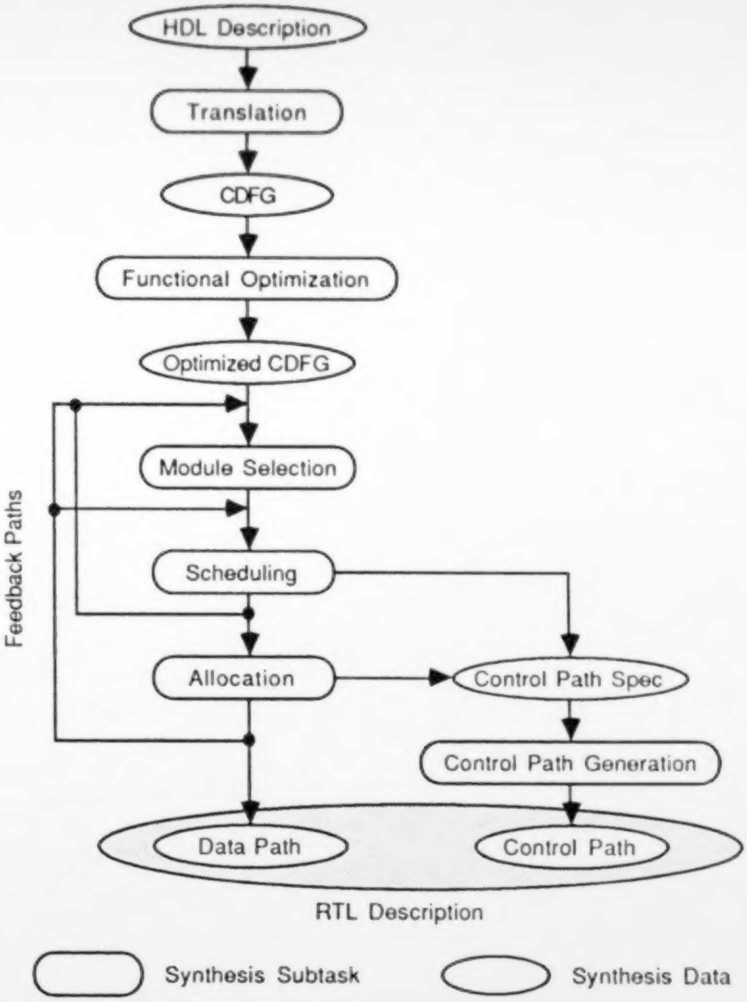

HDL Description
Translation
CDFG
Functional Optimization
Optimized CDFG
Module Selection
Scheduling
Allocation
Control Path Spec
Control Path Generation
Data Path
Control Path
RTL Description
Feedback Paths
Synthesis Subtask
Synthesis Data

Figure 2.1 Traditional Synthesis Process

the sequence of control steps in which the data value must be stored for subsequent use).

Given the lifetimes of operations and data values, the allocation step assigns cells to operations and data values, such that all operations (or data values) assigned to the same cell have non-overlapping lifetimes unless they are on *mutually exclusive* paths in the CDFG (see Chapter 7). This step also creates all interconnects required to transfer data between functional units and registers, thereby constructing a data path capable of implementing the scheduled CDFG.

After scheduling and allocation, the **control path generation** step constructs a FSM specification for the control path by mapping control steps to states, embedding precedence, loop and case constructs in the next-state logic, and specifying all control signals required in each control step as outputs in the corresponding state. This FSM specification is then optimized and passed to FSM generators for implementation.

However, a purely top-down design process suffers from the shortsightedness in estimating low-level costs for higher-level design decisions. Consequently, feedback paths are typically provided between module selection, scheduling and allocation as shown in Fig. 2.1. These feedback paths permit design iteration based on a stepwise refinement approach, in which the low-level design characteristics of each iteration are used to guide the high-level design decisions in the next iteration. Nevertheless, synthesis tasks are still performed separately, preventing a tight coupling of design decisions in these tasks.

2.2. Proposed Synthesis System

We propose here a framework in which different tasks in high level synthesis are solved concurrently. Basically, we organize all synthesis steps as concurrent processes in a blackboard architecture. The complete proposed system is depicted in Fig. 2.2. At the center of this system is a centralized data structure (i.e., the blackboard) and kernel software which provide access to all data *posted* on the blackboard. The primary data on the blackboard are the CDFG, the data path circuit structure, and the control path state graph and circuit structure.

In this system, each synthesis task is incrementally solved by an algorithmic

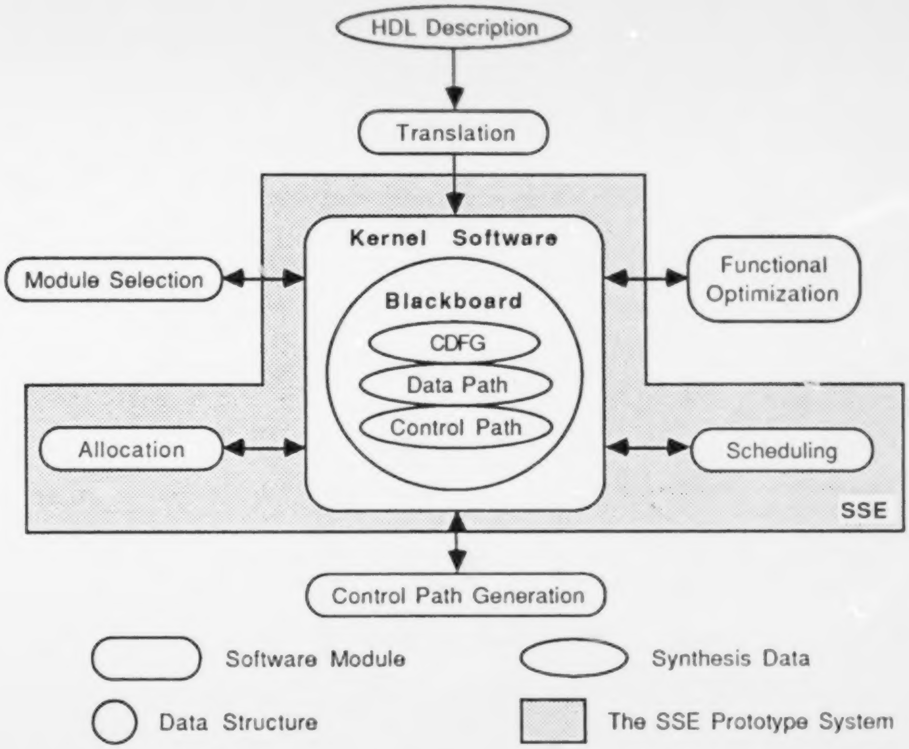

HDL Description
Translation
Kernel Software
Blackboard
CDFG
Data Path
Control Path
Module Selection
Functional Optimization
Allocation
Scheduling
SSE
Control Path Generation
Software Module
Synthesis Data
Data Structure
The SSE Prototype System

Figure 2.2 Complete Proposed System for Integrated High Level Synthesis

agent, which is a software module that queries and updates data on the blackboard. Each agent is required to work independently of other agents, and only makes one design decision, based on the current state of the blackboard, each time it is *activated*. Moreover, since there is no *a priori* order for the design steps, each synthesis algorithm must be able to accommodate (and hopefully, make use of) partial designs by other synthesis algorithms. For example, when the *scheduling agent* is activated, it may assign a control step to one unscheduled operation in the CDFG based on the partial schedules and allocations in the CDFG. Similarly, when the *functional optimization agent* is activated, it may migrate a small number of operations into a case construct based on the partial schedules in the CDFG. These synthesis agents are coordinated by a *supervisor* module which dynamically decides which agent should be activated next. This allows an arbitrary intermixing of synthesis steps.

Advantages of this synthesis system are threefold. First, this system allows a tight coupling between the design steps in different synthesis tasks, even though the synthesis agents are implemented independently of one another. Second, as a result of the independence between the synthesis agents, this system is well suited for experimenting with different synthesis algorithms. Synthesis agents may be easily added to or deleted from the system, requiring only updates to the supervisor module. Third, by arbitrarily intermixing synthesis steps, the supervisor module facilitates experimenting with different synthesis strategies. For example, the system can emulate other design approaches by enforcing different, fixed ordering of the synthesis steps.

On the other hand, in order to implement our proposed synthesis system, we must address two issues. First, we need new synthesis algorithms which are fast, incremental, and produce good designs. Second, we need to develop bottom-up design techniques in order to perform low level synthesis tasks (e.g., allocation) before higher level synthesis tasks (e.g., scheduling) are completed. In this thesis, we address these issues by focusing on integrating the tasks of scheduling and allocation, and we implement the prototype SSE system as a proof of concept.

2.3. Prototype System Overview

The prototype SSE system integrates the tasks of scheduling and allocation in high level synthesis as proposed in the previous section. This is depicted by the shaded area in Fig. 2.2.

2.3.1. Applying Simulated Evolution to Scheduling and Allocation

In the SSE system, we apply the technique of *simulated evolution* to the tasks of scheduling and allocation. Simulated evolution, or **SE**, is a general optimization technique modeled after the principle of natural selection [22]. Basically, given a CDFG, a greedy scheduling (or allocation) algorithm first schedules (allocates) all operations (and data values) in the CDFG using local cost functions. All scheduling (allocation) assignments are then graded by a global cost function and assigned normalized costs between 0 and 1. Subsequently, random thresholds between 0 and 1 are generated to determine the "survival" of these scheduling (allocation) assignments. For each assignment, if the randomly generated threshold is less than its normalized cost, then the corresponding operation (or data value) is de-scheduled (de-allocated). All de-scheduled operations (de-allocated operations and data values) are then re-scheduled (re-allocated) by the greedy algorithm in the next iteration. This implements a probabilistic hill climbing algorithm which is capable of escaping from local minima in the design space. Combined with fast greedy algorithms, SE-based synthesis algorithms generate very good designs in reasonable run times by rapidly searching over large regions of the design space [31, 32].

2.3.2. Allocation based on Fuzzy Schedules

While scheduling with partial allocation has been well researched, allocation with partial (i.e., incomplete) schedules has only recently received some attention [38, 54]. In the SSE system, we extend our allocation algorithm to accommodate incomplete schedules by considering unscheduled operations as having *uncertain* schedules. Basically, we know that each unscheduled operation will start in one of its candidate control steps, but we do not know which one until the operation is eventually scheduled. In the mean time, we may estimate the relative merit of the scheduling candidates and apply fuzzy set theory [24] to represent our *beliefs* that some control steps are better scheduling candidates than others. This leads to the concept of *fuzzy schedules* for CDFG operations. From these fuzzy schedules, we derive the concept of *fuzzy lifetimes* for all operations and data values, and then define a *fuzzy schedulability* which measures the belief that a *feasible* (i.e., conflict-free) schedule still exists given the allocation. This allows the allocation algorithm to handle incomplete schedules by trading off hardware cost and the *perceived risk of scheduling conflicts*. Consequently, we generalize the task of allocation to that of finding the smallest circuit for which a *feasible schedule is still highly possible.*

2.3.3. Integrating Scheduling and Allocation

In integrating the SE-based scheduling and allocation algorithms, we implement two mechanisms for the supervisor module to coordinate the synthesis algorithms. First, the supervisor dynamically decides whether greedy scheduling or greedy allocation (of a single operation or data value) should proceed next; and once all operations (and data values) are scheduled (allocated), the supervisor module decides whether to invoke probabilistic de-scheduling (de-allocation). Then the system will iterate by re-scheduling (re-allocating) all de-scheduled (de-allocated) operations (and data values). Second, the supervisor algorithm dynamically decides which operations and data values are subject to the probabilistic de-scheduling and de-allocation steps in the synthesis algorithms. This effectively focuses the attention of the synthesis algorithms on specific portions of the CDFG, and facilitates the implementation of more sophisticated synthesis strategies.

2.3.4. Assumptions

For the purpose of this thesis, we shall assume a synchronous hardware architecture in which a circuit consists of a data path and a control path. The data path is composed of functional units (e.g., adders, multipliers, ALUs), registers, multiplexers, busses and wires. The control path is a FSM, most likely micro-program or PLA-based, which supplies signals to cells in the data path to select functionalities in functional units, latch registers, setup multiplexer/bus addresses, etc. A simple two-phase clock scheme is assumed (see Fig. 2.3). In the first phase ($\bar{\phi}$), the interconnects are setup so that data from register outputs are transferred to the inputs of functional units, and data from functional unit outputs are transferred to the inputs of registers. The registers load their input data at the rising edge of the clock (unless loading is disabled). The control path derives its next state in the second phase (possibly depending on data path results produced in the first phase), and latches all control signals and state registers on the falling edge of the clock. Each clock period constitutes a *control step* because it corresponds to the duration of a state in the controller FSM. All data values are latched into registers in the first control step in which they become available. To simplify extraction of the control path FSM, the lifetime of a data value consists of those control steps in which a control signal (i.e., *load* or *hold*) must be generated for that data value. Consequently, a data value is *live* from the control step in which it is generated, until the control step just before it is last accessed.

Moreover, we shall assume that the inputs to the SSE system are a functionally optimized CDFG, a set of library cells for implementing all operations in the CDFG, and a global timing constraint which specifies the maximum number of control steps allowed for one execution of the CDFG. We assume that the hardware cost is calculated as the total area of cells (including interconnect cells) in the data path. While the cell areas of functional units and registers are specified by the cell library, the areas of

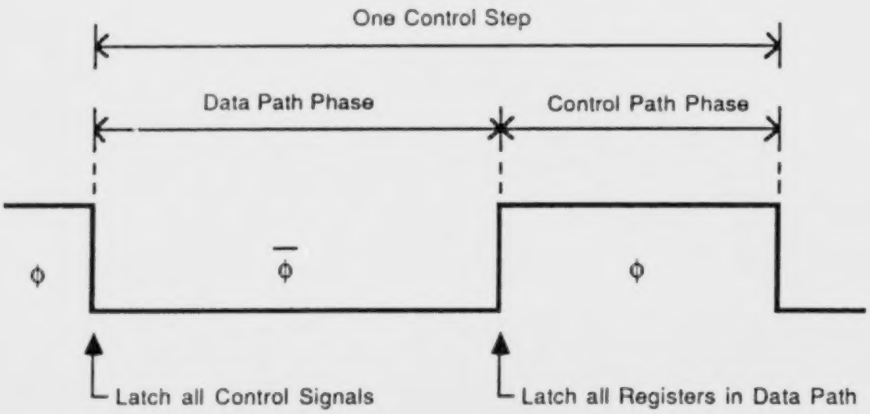

One Control Step
Data Path Phase
Control Path Phase
ϕ
$\bar{\phi}$
ϕ
Latch all Control Signals
Latch all Registers in Data Path

Figure 2.3 Two-Phase Clocking Scheme

multiplexers and busses are calculated by assuming that they are constructed from 2-input multiplexers. Specifically, we assume that busses are implemented as multiplexers with multiple fanouts, and each n-input multiplexer ($n > 2$) is implemented as $n - 1$ 2-input multiplexers. Consequently, we may express the total area of interconnects as the number of 2-input multiplexers required to implement all multiplexers and busses in the data path. Wiring area and propagation delays are ignored because their calculations require a circuit model at a much lower level of abstraction than is available prior to RTL synthesis and physical design.

2.3.5. Design Representations

Two major design representations in the SSE system are the CDFG (Control and Data Flow Graphs) and the CG (Circuit Graph). The CDFG specifies an algorithmic (or behavioral if partially allocated) description of a circuit, while the CG specifies the structural description of a circuit at the register-transfer level (i.e., interconnected functional units, registers and multiplexers, etc).

2.3.5.1. CDFG

The CDFG consists of two types of elements, namely *nodes* and *edges*. A CDFG node can be either a *data node*, which represents a *data path* operation (e.g., +, *, >), or a *control node*, which represents a control construct that is implemented in the *control path* (e.g., *loop*, *case*). Similarly, a CDFG edge can be either a *data edge*, which represents a data value generated by a data node, or a *control edge*, which represents a precedence or timing constraints on the scheduling of the CDFG. Scheduling and allocation information are recorded on CDFG nodes and edges as special attributes.

Data Node

A data node has a fixed number of data edge inputs and data edge outputs as determined by its functionality.

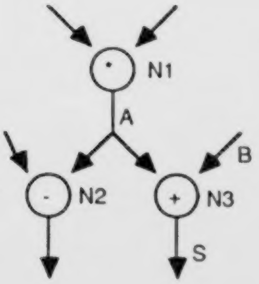


Figure 2.4 Sample CDFG Data Nodes and Data Edges

For example, the node $N3$ in Fig. 2.4 represents an integer addition operation with 2 inputs and 1 output (i.e., $S \leftarrow A + B$). This is an example of a commutative operation (i.e., $A+B = B+A$). If the selected cell for 2-input addition has permutable inputs (i.e., input ports may be exchanged), then $N3$ also has permutable inputs. As a result, we may exchange the data edges A and B at the inputs of this node during allocation (see Chapter 7).

Data Edge

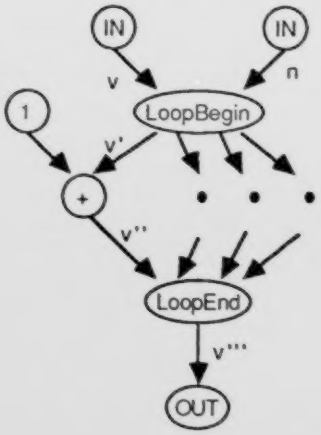
A data edge has a single input and multiple outputs. The node connected to the input of an edge is referred to as the *producer*, or *source*, node, and the nodes connected to the outputs of an edge are referred to as the *consumer*, or *sink*, nodes. In Fig. 2.4, the data edge A has a producer node labeled $N1$, and consumer nodes labeled $N2$ and $N3$. In this case, we say that the edge A is *produced* by the node $N1$, and *accessed* by the nodes $N2$ and $N3$.

Control Node

A control node has an arbitrary number of inputs and outputs.

```
READ(v);  READ(n);
LOOP for n times
  begin
     ...
     v <- v + 1;
     ...
  end LOOP;
WRITE(V);
```

(a)



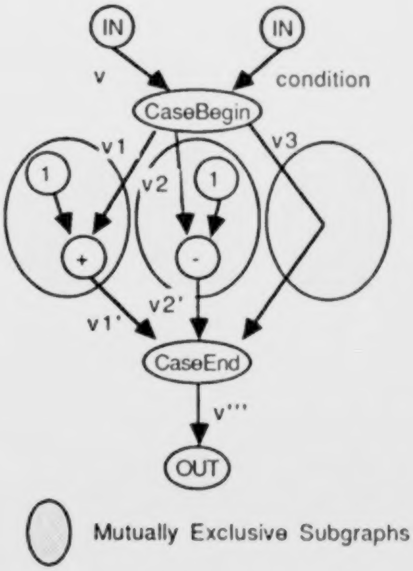
(b)

Figure 2.5 Sample HDL and CDFG with a LOOP Construct

Fig. 2.5(a) shows a HDL code for a simple loop, and Fig. 2.5(b) shows the corresponding CDFG loop construct, which consists of a *loop begin* node, a loop body (CDFG subgraph), and a *loop end* node. In this example, a variable v is assigned a value outside the loop, accessed and then modified inside the loop body, and then accessed by an operation after the loop. The four edges labeled v, v', v'' and v''' in Fig. 2.5(b) represent the same logical data, and must be allocated to the same register (at least for just before and just after the loop, and at the beginning/end of all loop iterations) in order to ensure functional correctness regardless of the value of n (i.e., the numbers of iterations in the loop). Consequently, corresponding loop begin and loop end nodes impose certain allocation constraints on some CDFG edges.

```
READ(v);  READ(condition);
CASE condition is:
  begin
    cond1:  v <- v + 1;
    cond2:  v <- v -  1;
    otherwise:
  end CASE;
WRITE(v);
```

(a)



(b)

Figure 2.6 Sample HDL and CDFG with a CASE Construct

Fig. 2.6(a) shows a HDL code for a simple case statement, and Fig. 2.6(b) shows the corresponding CDFG case construct, which consists of a *conditional fork* node, a set of parallel CDFG subgraphs, and a *join* node. Each of the parallel CDFG subgraphs enclosed by the fork and join nodes represents a *conditional branch* in the case construct. By definition of the case statements, in each pass of the CDFG one and only one of the branch conditions can be true. Consequently, operations and data values in separate branches are ***mutually exclusive***, in that they will never be live simultaneously even if they are scheduled to the same control steps. Again, the edges labeled v, $v1$, $v1'$, $v2$, $v2'$, $v3$, and v''' represent the same logical data, and must be allocated to the same register (at least for just before and just after the case construct) in order to ensure correctness regardless of which branch of the case construct is taken during execution.

Control Edge

A control edge may have multiple inputs and multiple outputs.

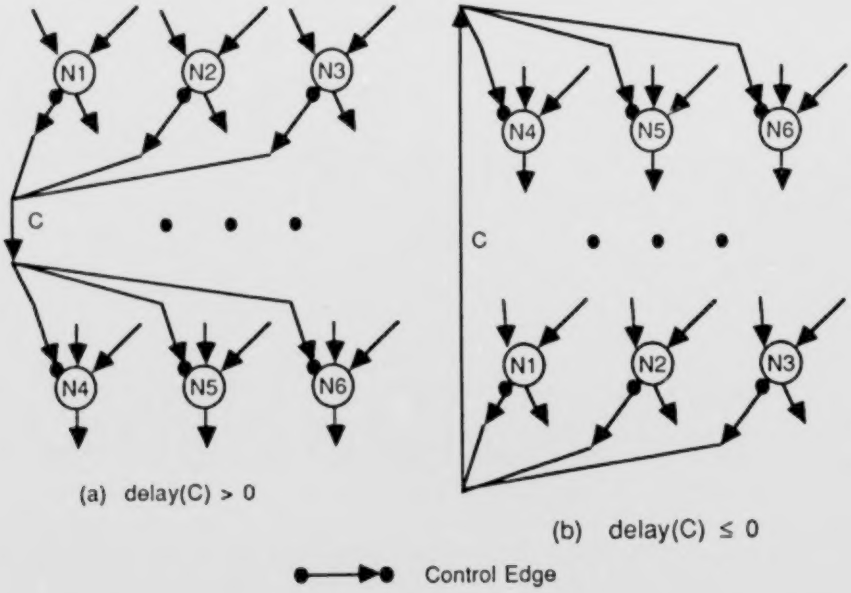


Figure 2.7 Example of (a) Minimum and (b) Maximum Timing Constraints

Fig. 2.7 shows a control edge labeled C specified from the nodes $N1$, $N2$ and $N3$ to the nodes $N4$, $N5$ and $N6$. The weight on the control edge ($delay(C)$ in Fig. 2.7) specifies a ***minimum*** delay constraint (in number of control steps) between (the starting control steps of) the producer nodes and (the starting control steps of) the consumer nodes of the control edge. If $delay(C) > 0$ (Fig. 2.7(a)), then C represents a local minimum timing constraint between the two groups of nodes (i.e., the latest of $N1$, $N2$, $N3$ must start at least $delay(C)$ control steps before the earliest of $N4$, $N5$, $N6$ starts). If $delay(C) < 0$ (Fig. 2.7(b)), then C represents a local maximum timing constraint between the two groups of nodes (i.e., the latest of $N1$, $N2$, $N3$ must start no later than $delay(C)$ control steps after the earliest of $N4$, $N5$, $N6$ starts). By convention [26], the case $delay(C) = 0$ is assumed to specify local maximum timing constraints (i.e., treated as if 0 is a negative number).

2.3.5.2. CG

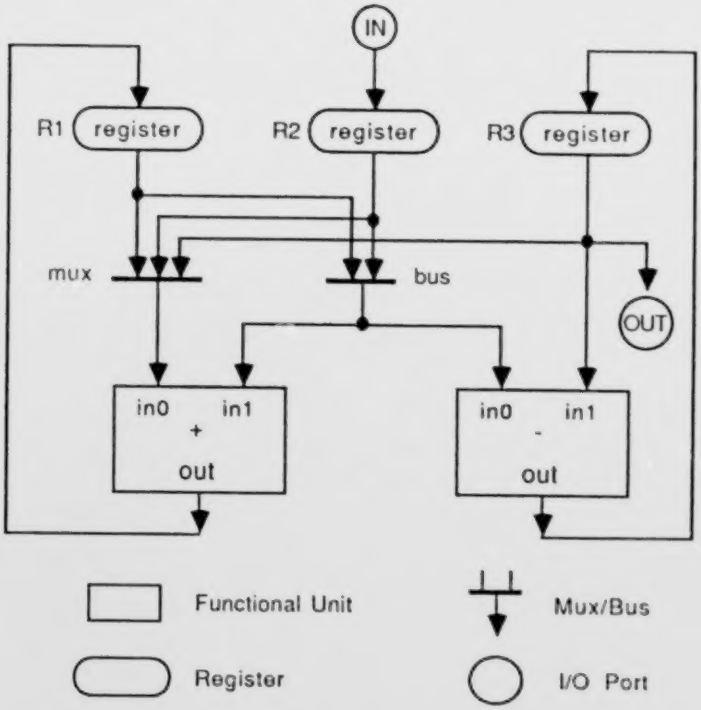

IN
R1
register
R2
register
R3
register
mux
bus
OUT
in0
in1
+
out
in0
in1
-
out
Functional Unit
Mux/Bus
Register
I/O Port

Figure 2.8 A Sample CG with 2 Functional Units, 3 Registers, a Mux and a Bus

The CG contains two types of elements: cell instances and signal nets. A cell instance (or simple *cell*) represents an instantiation of a library cell in the data path, while a signal net (or simply *net*) represents a circuit connection between one output signal of a cell and multiple input signals of other cells. To connect multiple nets to an input port of a cell, a multiplexer is created to select between the nets in each control step as required by the operations allocated to the cell. Busses are modeled as multiplexers whose outputs have multiple fanouts. Fig. 2.8 shows a simple example of the CG.

Chapter 3. Related Work

Existing systems for high level synthesis can be classified into three categories by the way they approach the tasks of scheduling and allocation. The first group of systems performs scheduling and allocation separately. This includes Facet [57], EMUCS [15], Chippe [3], HAL [47] and BUD [38]. The second group of systems performs scheduling and allocation concurrently. This group consists of Elf [11], MAHA [46], and SSE. The third group of systems does not directly perform the tasks of scheduling and allocation, but iteratively apply design transformations which may change either the schedules, the allocations, or both, to improve the initial designs. This includes DAA [25], Hercules [5], and CAMAD [49].

In this chapter, we give a brief overview of these synthesis systems, and compare different synthesis approaches. Interested readers are referred to [37] for a comprehensive review of high level synthesis systems and approaches.

3.1. Separate Scheduling and Allocation Systems

Most of the existing synthesis systems perform scheduling and allocation separately to simplify the problems. For the most part, scheduling is performed prior to allocation, although recent work [38,54] has perform *pre-allocation*, in which a CDFG is partitioned based on low level estimates (e.g., connectivities, propagation delay and area), before scheduling and final allocation.

Facet

Facet uses *as soon as possible* scheduling, in which operations are assigned to their earliest candidate control steps according to data dependencies (i.e., an operation cannot start unless all of its inputs are available). It then formulates the allocation of operations, registers and data transfers as separate clique partitioning problems. For example, to allocate operations to functional units, a graph is created from all operations by connecting two operations with an arc if they may be allocated to the same functional unit (i.e., their lifetimes do not overlap). By solving for the minimum number of cliques in this graph, and assigning all operations in each clique to the same functional unit, one can allocate operations to the minimum number of functional units. Unfortunately, minimizing the number of functional units, registers and interconnects separately does not produce an allocation which minimizes the total hardware cost.

EMUCS

EMUCS makes use of a simple list scheduling algorithm [37] to schedule operations. It then incrementally allocates operations to functional units, data values to registers and data transfers to interconnects. The allocation algorithm successively selects the most critical allocation to perform, which is defined as the best (i.e., minimum cost) allocation for an operation (or data value) which has the largest cost difference with respect to the second best allocation for the same operation (data value). A partial circuit is incrementally updated as allocation decisions are made, to facilitate cost calculations for subsequent allocation candidates.

Chippe

Chippe uses the Slicer module for scheduling and the Splicer module for allocation. Slicer [42] is based on the list scheduling algorithm, which levelizes all operations in the CDFG (from top to bottom), and then schedules operations in increasing levels, delaying operations whenever the maximum number of functional units are reached in a control step. Splicer [43] incrementally allocates operations and data values, starting from operations scheduled in the first control step. Splicer makes use of four special heuristics when allocating data transfers in order to reduce interconnect costs. These heuristics explicitly consider interconnects between 1) register outputs and bus inputs, 2) bus outputs and functional unit inputs, 3) functional outputs to bus inputs, and 4) bus outputs to register inputs. Chippe incorporates a knowledge-based system which adjusts parameters of Slicer and Splicer based on design statistics. This implements a *design critic* which tunes the synthesis algorithms in successive iterations to improve the designs.

BUD

BUD first partitions a CDFG according to a distance metric between each pair of operations which considers their functionality, degree of connections, and potential parallelism. It then schedules operations using a list scheduling algorithm which gives priority to operations on the critical paths in the CDFG. Allocation is then performed on operations and data values in each partition separately, and a floorplan is developed to evaluate area and delay costs of the design. This design is then passed to the DAA synthesis system for further optimization.

HAL

HAL performs scheduling using a *force directed scheduling* algorithm which successively makes the best scheduling decisions based on a global calculation of attraction and repulsion forces between operations. This tries to minimize the hardware cost of a schedule by balancing the numbers of concurrent operations, data values and data transfers across control steps. After scheduling, HAL uses greedy algorithms to first allocate operations to functional units, then data values to registers, and finally data transfers to interconnects. These preliminary allocations are then optimized by first exhaustively trying all permutations of functional unit inputs for commutative operations, and then merging registers and multiplexers using weight-directed clique-partitioning algorithms.

3.2. Concurrent Scheduling and Allocation Systems

Scheduling and allocation are highly interdependent tasks. Elf and MAHA attempt to exploit this interdependency by performing scheduling and allocation concurrently, although in both cases allocation is performed as a secondary task which guides the scheduling algorithm.

Elf

Elf uses a list scheduling algorithm which gives priority to the operation with the maximum "urgency", which is defined as a weighted number of control steps between the candidate control step and the "as late as possible" schedule for an operation. Elf also considers potential allocation costs for different scheduling decisions. Each time an operation is scheduled, it is immediately allocated to a functional unit. This incrementally builds up a partial circuit, which is then used to estimate allocation costs associated with different scheduling candidates. Intuitively, if the allocation cost is high, Elf would delay the schedule of an operation until its urgency outweighs its allocation cost.

MAHA

MAHA starts by sequentially allocating operations on the critical path and constructing a partial circuit. Operations off the critical path are then iteratively scheduled in order of increasing "freedom", which is defined as the number of candidate control steps. When scheduling an operation off the critical path, existing functional units are examined for possible hardware sharing. The first control step for which possible hardware sharing is found is scheduled to the operation. If no hardware sharing is possible, the operation is scheduled to its "as soon as possible" control step. Again, operations are allocated as soon as they are scheduled to update the partial circuit for subsequent scheduling steps. After scheduling (and allocation of operations), the REAL system [28] is used to allocate data values to registers. REAL uses the *left-edge* algorithm to pack data values with disjoint lifetimes to the same registers to minimize the number of registers.

3.3. Transformational Synthesis Systems

Transformational synthesis systems start with a default design, which is typically the maximally parallel or maximally serial design, and then successively apply design transformations to improve the current design.

DAA

DAA is a rule-based expert system which improves a design by applying transformations which directly trade off fast, costly parallel circuitry against slower, smaller sequential circuitry. DAA originally used a maximally parallel design (i.e., each operation is allocated to a cell by itself), but now uses the output from BUD as the initial design.

Hercules

Hercules first generates a default schedule and then a preliminary allocation by assuming zero delays for combinational circuits (i.e., a maximally serial design). It then applies logic synthesis to the design to optimize combinational circuitry and obtain area and timing estimates for this circuitry. Iterative refinement is then performed in which the initial schedules are modified based on information fed back from logic synthesis. In this case, data path transformations are embedded in the logic synthesis system, while control path transformations are guided by area and delay estimates provided by logic synthesis.

CAMAD

CAMAD also starts with a maximally parallel design, and then carries out iterative refinement by applying optimizing transformations which directly trade off circuit area and timing performance. A two level optimization strategy has been implemented in CAMAD: for operations on the critical paths, CAMAD selects transformations which minimize timing delay; for operations off the critical paths, CAMAD selects transformations which minimize circuit area.

Devadas' System

Devadas' system [6] formulates the scheduling and allocation problem as a single, two-dimensional (i.e., space and time) *placement* problem, and applies the general optimization technique of simulated annealing [21] to this problem. Starting with a randomly generated design, this synthesis system randomly applies one of a small number of design transformations, or *moves*. If the new design is better than the current one, then the new design is accepted as the current design; otherwise, the new design is probabilistically accepted, depending on the cost increment and a control parameter, the "temperature". Allocation of data transfers to interconnects is not explicitly modeled because it does not easily fit into the two-dimensional placement formulation. Instead, an estimate for the number of busses required is used to calculate the interconnect costs for different designs.

3.4. Comparison

Among all of the systems described above, HAL produces the smallest circuits to date for data path synthesis. This is to be expected because HAL is the only system using global scheduling (i.e., HAL calculates the cost of *all* candidate control steps for *every* unscheduled node to determine which node to schedule next), which should produce better results than local scheduling algorithms (i.e., algorithms which pick the next node to schedule without comparing all candidate control steps for every node). However, since the complexity of HAL's force-directed scheduling algorithm increases rapidly with the number of operations and the degree of scheduling freedom, HAL is limited to designs with tight timing constraints. For designs with loose timing constraints, local algorithms (e.g., force directed list scheduling, or FDLS [47]) must be used to reduce the computational complexity.

From the point of view of technology potentials, synthesis systems which perform concurrent scheduling and allocation are better than systems which perform these tasks separately due to the heavy interdependency between these tasks. However, the computational complexity due to concurrent scheduling and allocation is such that greedy algorithms and local heuristics must be used. Consequently, these systems tend to get trapped in local minima, and require many design iterations to produce reasonable circuits. To date, no system using concurrent scheduling and allocation has been able to compete with HAL in circuit quality because the computational costs for design iterations are prohibitive.

BUD and Hercules are interesting synthesis systems primarily because they represent the first steps in integrating high level synthesis with physical design tools. Just as concurrent scheduling and allocation is potentially better than separate scheduling and allocation, integrating physical design tools, such as floorplanning, module compilers and logic synthesis tools, with high level synthesis is important for producing good designs at the final layout level. Unfortunately, the same issue of computational complexity must be resolved before these systems can compete with other systems which perform synthesis tasks in isolation.

Transformational synthesis systems are difficult to compare with conventional high level synthesis systems because they are often targeted for specific application domains, such as digital signal processing (DSP) and microprocessor designs, for which large amounts of expert knowledge is available. Such application specific design knowledge are encoded in the forms of design transformations and the heuristics which govern the application of these transformations. While a well tuned transformational synthesis system may produce designs which are comparable to human designers in their target domain, major effort is required to port such a system to another application domain. Nevertheless, a significant advantage of transformational synthesis systems is that, once their transformations have been formally verified, they generate verifiable designs which are correct by construction.

Recent work in applying transformational synthesis to general purpose synthesis applications, such as Devadas' system, percolation scheduling [50] and SALSA [40], uses a small set of design transformations and simple heuristics for applying these transformations. These systems rely on probabilistic search and rapid design iterations to achieve good designs.

Depending on the supervisor algorithm used, the SSE system may be classified into any one of the three groups of synthesis systems in the previous section. If we consider SSE as performing scheduling and allocation separately, then it differs from previous synthesis systems in this group in the way SE-based synthesis algorithms combine simple, fast greedy algorithms and probabilistic hill climbing to produce comparable designs very quickly, and much better designs when given longer run times (see Chapter 8).

If we consider SSE as performing scheduling and allocation concurrently, then it differs from previous synthesis systems in this group in the way it allows arbitrary intermixing of scheduling and allocation steps. In particular, SSE allows allocation of unscheduled operations and data values whose lifetimes are ill-defined (see Chapter 10), unlike Elf and MAHA which only allocate operations and data values with well-defined lifetimes.

Finally, SSE may be considered as performing transformational synthesis in which each design iteration corresponds to the application of a complex design transformation. In this case, SSE differs from previous transformational synthesis systems in the way it generates arbitrarily complex transformations *on the fly* depending on cost functions in the rip-out and re-construction steps in SE-based synthesis.

Chapter 4. Optimization by Simulated Evolution

Simulated Evolution, or SE, is a general optimization technique based on the analogy between optimization and evolutionary processes. It has been successfully applied to such problems as standard cell placement [22], switch box routing [29], and circuit partitioning [52]. In this chapter, we will first introduce the general SE algorithm, and then describe our implementation of SE in the context of an *optimum assignment problem.* This separates all application specific components of the SE algorithm into three cost functions, namely *PRIORITY*, *INCR* and *GLOBAL*, and paves the way for the definition of SE-based scheduling and allocation algorithms in Chapters 5 and 6.

4.1. The SE Algorithm

Evolution has two essential features: namely *hereditary variation* and *differential reproduction.* Hereditary variation requires that an evolutionary system changes through time such that each new state is similar to the current state. On the other hand, differential reproduction requires that an evolutionary system be subjected to an evaluation process and probabilistically discards the inferior parts of the system and retains the superior parts for subsequent states. The most notable example of evolutionary processes is that of biological evolution, in which genetic inheritance and natural selection combine to determine the most desirable characteristics of a population.

Given a combinatorial optimization problem specified by a finite set of solutions, S, and a cost function $C(X)$ defined on all solutions $X \in S$, the SE algorithm (Fig. 4.1) searches for the optimum solutions in S by simulating an *evolution of solutions.* At each state in this simulated evolution, SE generates a new solution from the current solution in two steps: *SELECT* and *GENERATE*. The *SELECT* step simulates differential reproduction. In this step, all elements in the current solution are evaluated with respect to their contributions to the overall cost, C, of the solution. Then elements with high costs are probabilistically removed from the solution, and the remaining elements are returned as a *partial solution*. This partial solution is passed to the *GENERATE* step, which simulates hereditary variation. In this step, a new solution is obtained from the partial solution by adding the missing elements to the partial solution, thus retaining a similarity with the current solution and introducing variations at the same time. Fig. 4.1 gives the pseudo-code for the SE algorithm.

```
Algorithm SE;
  /* X is a complete solution */
  /* $X_{best}$ is the best solution found */
  /* p is a partial solution */
  begin
    X := GENERATE(∅);
    $X_{best}$ := X;
    loop until TERMINATE();
      begin
        p := SELECT(X);
        X := GENERATE(p);
        if (C(X) < C($X_{best}$)) then $X_{best}$ := X;
      end;
    return($X_{best}$);
  end SE;
```

Figure 4.1 Pseudo-Code for the SE Algorithm

Initially, SE constructs the first solution from the ground up by applying the function *GENERATE* to the empty partial solution, ∅. Subsequently, the algorithm repeatedly alternates between *SELECT* and *GENERATE*, producing a new solution from the current solution in each iteration. Throughout these iterations, the minimum cost solution is recorded, and eventually returned as the best solution found by SE. The termination criteria, implemented by *TERMINATE*, are implementation dependent. However, two useful termination criteria are when a satisfactory solution is found, and when the number of SE iterations reaches a user-specified maximum.

Since *SELECT* is probabilistic, and since the results of *SELECT* and *GENERATE* depend only on the current solution, the evolution of solutions in SE is a *stochastic process* [20] in which each state (i.e., solution) depends only on its previous state. Consequently, the SE algorithm may be modeled by a *Markov chain with stationary transitions* [7]. A useful result of Markov chains is that, if a Markov chain is *irreducible* (i.e., every state i can reach every other state j in a finite number of steps), *aperiodic* (i.e., every state has 1 as the greatest common denominator of all numbers of steps in which this state can reach itself) and *recurrent* (i.e., every state can reach itself in a finite number of steps with probability 1), then there exists a positive *stationary probability*, π_i, for each state i in the Markov chain. The existence of these stationary probabilities implies that, given sufficiently long time, the probability of the Markov chain being in any state i is π_i, and is independent of the initial state of the Markov chain. In other words, if we can implement SE in such a way that the existence of stationary probabilities is guaranteed, then we can be assured that, independent of the initial solution (i.e., *GENERATE*$(\emptyset)$), the SE algorithm asymptotically obtains the optimum solution.

We propose two conditions which guarantee the existence of stationary probabilities for SE. First, given any solution in S, we require *SELECT* to have a non-zero probability of returning $\emptyset$ as the partial solution. Second, given any partial solution, we require *GENERATE* to have a non-zero probability of returning any solution in S as the complete solution. Basically, these two conditions guarantee a positive transition probability between every pair of solutions in S. It is then straightforward to show that these are *sufficient* conditions for SE to be *irreducible*, *aperiodic*, and *recurrent*, and hence for SE to asymptotically obtain the global optimum independent of its initial solutions.

The organization of this chapter is as follows. In the next section, we will define an *optimum assignment problem* which is a useful abstraction of the tasks of scheduling and allocation. Given this generic problem, Sections 4.3 and 4.4 describe our implementation of *GENERATE* and *SELECT*, respectively, in accordance with the above conditions for the desired asymptotic behavior of SE. Finally, Section 4.5 summarizes the cost functions which fully characterize our application of the SE algorithm to any combinatorial optimization problems which may be formulated as optimum assignment problems.

4.2. Optimum Assignment Problem

In the *optimum assignment problem,* we wish to assign a finite set of values to N variables so as to minimize a cost function, C, subject to certain constraints, Ω. Many design problems, including the synthesis tasks of scheduling and allocation, may be formulated as such problems. In particular, we are interested in the class of NP-hard problems [8] for which all known exact algorithms have computational complexities which increase exponentially with the problem size, N.

More formally, suppose there is a set of N variables, denoted by $V = \{v_1, v_2, \ldots, v_N\}$, and a finite set R of values. Let $b_i = (v_i , r)$ $(i = 1 ,\ldots, N)$ denote the binding, or *assignment,* of a value $r \in R$ to the variable v_i. Then we may define a solution, X, as a *set* of N assignments:

$$X = \{ b_i : i = 1 , 2 ,\ldots, N \},$$

and define the solution space, S, as the *set* of all possible combinations of N value assignments:

$$S = R \times R \times \cdots \times R = R^N .$$

Moreover, given the constraints Ω, we say a solution, X, is *feasible* if all value assignments in X satisfy Ω, otherwise X is *infeasible*. Let $F \subseteq S$ denote the set of all feasible solutions. Then the objective of the optimum assignment problem is to find a feasible solution, $X \in F$, which minimizes the objective cost, $C(X)$.

In general, the cost function C is only well defined on the set of feasible solutions, F. For the purpose of this thesis, however, we shall assume C is well defined on S since we permit SE to generate infeasible solutions in S. This involves no real loss in generality, since we can easily modify any cost function to be well defined on infeasible solutions by adding high penalty costs for any violation of Ω so that, for each pair of solutions $i \in F$ and $j \notin F$, $C(i) < C(j)$ is guaranteed.

Finally, we define a partial solution, p, as a set of N or fewer assignments:

$$p \subseteq \{ b_i : i = 1, 2, \ldots, N \}.$$

Given such a partial solution, p, we say that a variable v_i is *unassigned* in p if $b_i \notin p$, that v_i is *de-assigned* from p if b_i is removed from the set p, and that v_i is *assigned* a value r if $b_i = (v_i, r)$ is added to p.

4.3. Implementation of GENERATE

```
Procedure GENERATE(p);
  begin
    M := get all unassigned variables in p;
    X := p;
    while (M is not empty) do
      begin
        v := select a variable in M with maximum PRIORITY(v);
        if (RANDOM(0,1) < $\mu$)
          then r := MUTATION(v);
          else r := select a value for v with minimum INCR(v,r);
        remove v from M;
        assign r to variable v;
        update X;
      end;
    return(X);
  end;
```

Figure 4.2 Pseudo-Code for GENERATE

The function *GENERATE* takes a partial solution, p, as input, and produces a complete solution, $X \supseteq p$, as output. To guarantee that SE asymptotically obtains the global optimum, we require *GENERATE* to be able to produce every solution, $X \in S$, with a positive probability when given the partial solution $p = \emptyset$. Trading off solution quality for computational speed, we implement *GENERATE* using a greedy algorithm with a random mutation operator. Fig. 4.2 gives the pseudo-code for our implementation of *GENERATE*.

Given a partial solution, p, *GENERATE* first collects all unassigned variables in p into a set M, and then iteratively assigns a value to the variable, v, in M with the highest priority (i.e., *PRIORITY*(v)). For each unassigned variable v, if a randomly generated number between 0 and 1 (i.e., *RANDOM*(0,1)) is less than a constant μ (the mutation probability), then a mutation operator is invoked which randomly assigns a value for v. Otherwise (i.e., *RANDOM*$(0,1) \geq \mu$), the algorithm evaluates all candidate values for v, and assigns to v the value r with the lowest incremental cost, *INCR*(v,r).

The mutation probability, μ, ensures that *GENERATE* is capable of producing every possible solution when given $p = \emptyset$. In practice, μ is assigned a small number (e.g., 0.01) so that, for the most part, *GENERATE* relies on heuristics implemented in *PRIORITY* and *INCR* to assign values to variables.

4.4. Implementation of SELECT

The function *SELECT* takes a complete solution, X, as input, and produces a partial solution, $p \subseteq X$, as output. To simulate differential reproduction, *SELECT* must evaluate the assignments in X, and then randomly remove each $b_i \in X$ from X (i.e., de-assign the variable v_i) with a probability which increases as the cost of b_i increases. Moreover, to guarantee that SE asymptotically obtains the global optimum, we require *SELECT* to be able to produce the partial solution $p = \emptyset$ with a positive probability when given any complete solution $X \in S$. Fig. 4.3 shows the pseudo-code for our implementation of *SELECT*.

```
Procedure Select (X);
 begin
   p := X;
   for all assignments b=(v,r) do calculate GLOBAL(b);
   min := get minimum of GLOBAL(b);
   max := get maximum of GLOBAL(b);
   for all assignment b=(v,r) do
     begin
       norm := (GLOBAL(b) - min) / (max - min);
       if (RANDOM($-\delta$,1) < norm) then
         begin
           de-assign v from p;
           update p;
         end;
     end;
   return(p);
 end;
```

Figure 4.3 Pseudo-Code for SELECT

Given a solution, X, *SELECT* first calculates a cost, $GLOBAL(b)$, for each value assignment $b = (v, r)$ in X, normalizes these values to be between 0 and 1 inclusive, and then probabilistically removes each value assignment b based on its normalized cost, *norm*. For each value assignment, $b = (v, r)$, if a randomly generated threshold cost between $-\delta$ and 1 (i.e., $RANDOM(-\delta,1)$) is less than its normalized cost, *norm*, then the algorithm removes b from p (i.e., de-assigns the variable v from p). This probabilistically removes each assignment $b \in X$ based on the (normalized) value of $GLOBAL(b)$. In the end, the remaining assignments are returned in the partial solution, p.

The constant δ is a small positive number (e.g., 0.01) which ensures that the probability of any value assignment being removed is at least $\frac{\delta}{1+\delta} \approx \delta$ (assuming $RANDOM(-\delta,1)$ has a uniform probability distribution). Since the probability for removing all value assignments in any solution $X \in S$ is at least (approximately) δ^N, keeping

$\delta > 0$ ensures that *SELECT* will produce the partial solution $p = \varnothing$ from any solution with a positive probability.

4.5. Summary

Based on the above implementation of *GENERATE* and *SELECT*, we may characterize our application of the SE algorithm to any combinatorial optimization problem by first formulating it as an optimum assignment problem, and then specifying the cost functions *PRIORITY*, *INCR* and *GLOBAL*. The function *PRIORITY* determines the next unassigned variable to be processed by *GENERATE*. This implements application specific heuristics to order the assignments of interdependent variables. The function *INCR* determines the value assigned to each unassigned variable by *GENERATE* when the mutation operator is not invoked. This estimates the incremental increase in the objective cost, C, due to each candidate assignment. The function *GLOBAL* determines the probability with which a value assignment is randomly removed from a solution by *SELECT*. This calculates the pro-rated cost of each value assignment based on its contribution to the overall cost of the solution.

The cost functions *PRIORITY*, *INCR* and *GLOBAL* are important in that they significantly affect the performance of the SE algorithm. However, they do not affect the *correctness* of SE, since our *SELECT* and *GENERATE* functions ensure that SE asymptotically obtains the global optimum regardless of these cost functions.

Chapter 5. Basic Scheduling Algorithm

In this chapter, we will first review previous scheduling approaches, and then describe our application of SE to the scheduling task. To do so, we will first formulate the scheduling task as an optimum assignment problem, and then characterize our SE-based scheduling algorithm by defining the cost functions *PRIORITY*, *INCR* and *GLOBAL*.

To simplify the discussions in this chapter (and those in Chapter 6), we shall temporarily assume that the CDFG contains only data nodes and data edges, and that all CDFG nodes are scheduled prior to allocation. The first assumption will be removed in Chapter 7, which describes extensions to the basic scheduling and allocation algorithms for loop and case constructs, local timing constraints, and other synthesis features such as pipelining, chaining, and interconnect optimization. The second assumption will be removed in Chapter 10, which describes extensions for allocating a partially scheduled CDFG and for scheduling a partially allocated CDFG.

5.1. Previous Scheduling Approaches

Existing scheduling algorithms can be classified into two groups: namely the iterative/constructive group and the transformational group [37]. An iterative/constructive scheduling algorithm successively assigns operations to control steps until a complete schedule is constructed. Different algorithms in this group differ in their heuristics for ordering operations to be scheduled and for selecting which candidate control step is scheduled to each operation. The most basic of these scheduling algorithms is the *as soon as possible* (ASAP) scheduling [15,56,60], in which the operations are ordered by their data dependencies, and each operation is assigned to the earliest control step possible, subject to hardware resource constraints. *List scheduling* [11,41] improves on the ASAP scheduling by using a more global view on ordering operations and selecting control steps for these operations. Starting from the first (last) control step, list scheduling collects all operations whose predecessors (descendents) have all been scheduled in previous control steps into a list of candidate operations. These candidate operations are then sorted by some priority function, which is usually related to how critical or how constrained an operation is. Each operation is then considered in turn, and may be either scheduled in the current control step or deferred to subsequent control steps depending on hardware resource constraints.

Global formulations of the scheduling problem include freedom-based scheduling [46], force-directed scheduling [47], and integer linear programming based scheduling [9, 18]. In freedom-based scheduling, the operation with the least *freedom*, which is defined as the number of candidate control steps for that operation, is scheduled first. This gives priority to operations on the critical paths, and then to operations less and less critical in terms of timing constraints. Operations with equal freedoms are scheduled in arbitrary order. In force-directed scheduling, attraction and repulsion forces are calculated for each of the candidate control steps for each operation. The operation/control step pair with the smallest net force is selected for scheduling. All forces are then updated to reflect this scheduling assignment. In ILP-based scheduling (i.e., scheduling based on integer linear programming), the scheduling problem is translated to a system of linear inequalities, which are then solved using integer linear programming techniques (i.e., branch and bound).

Most of the above scheduling algorithms (i.e., other than those based on ILP which uses branch and bound) are ultimately based on greedy algorithms. As such they are vulnerable to local minima in the design space. While increased look-aheads and complex cost functions have improved the quality of these scheduling algorithms to some degree, they do not address the fundamental problem of poor design space exploration in greedy algorithms.

On the other hand, transformational scheduling algorithms depend on effective exploration of the design space to obtain good designs. Transformational scheduling algorithms start with a default schedule, which is typically either the maximally serial or the maximally parallel schedule, and then repeatedly apply correctness preserving design transformations to improve the current schedules. Different scheduling systems in this group differ in their suite of transformations and in how they select between applicable transformations. The Yorktown Silicon Compiler [4] and the CAMAD [49] systems are based on transformational scheduling, in which large numbers of expert system rules constitute the design transformations. In percolation scheduling [50], a small number of primitive transformations, or *moves*, are defined, and a host of heuristics and cost functions are used to guide the application of these moves. Simulated-annealing-based (SA) scheduling [6, 53] also contains a small number of primitive moves, but randomly applies these moves to the current schedules, and probabilistically accepts the new schedules in its search for the globally optimum schedule.

Unlike iterative/constructive scheduling algorithms, most transformational scheduling systems can escape from local minima in the design space because they incorporate *hill climbing* moves (i.e., the new schedules may be inferior to the current schedules). Therefore transformational scheduling has the potential of obtaining the global optimum through effective design space exploration. However, the performance of such scheduling systems depends greatly on the power of their transformations, and on the order in which these transformations are applied.

As it turns out, applying SE to the scheduling problem combines the advantages of both iterative/constructive scheduling and transformation scheduling. Basically, SE-based scheduling explores the design space by repeatedly ripping out parts of schedules and then re-constructing these parts to produce new schedules. This is simply iterative/constructive scheduling with design iterations, in which hill climbing moves are allowed because the re-construction steps may produce worse schedules due to the simplistic and greedy heuristics in *PRIORITY* and *INCR*. On the other hand, SE-based scheduling can also be seen as transformational scheduling, in which arbitrarily complex design transformations are generated *on the fly* depending on cost functions and heuristics in the rip-out and re-construction steps. Consequently, SE-based scheduling is simple, fast, and produces good schedules by its effective exploration of the design space (see Chapter 8).

5.2. Scheduling as an Optimum Assignment Problem

Given a CDFG containing N nodes, denoted by n_i $(i = 1, 2, ..., N)$, and a maximum global timing constraint of G control steps (G is a positive integer), we formulate the task of scheduling as an optimum assignment problem as follows. Let s_i $(i = 1, 2, ..., N)$ denote the starting control step of the node n_i, and let CS denote a finite set of control step numbers from 1 to G (i.e., $CS = \{1, 2, ..., G\}$). We define the assignment $b_i = (s_i, t)$ $(i = 1, 2, ..., N$ and $t \in CS)$ as *scheduling* the node n_i to start in control step t. Then we define a *complete schedule*, denoted by $SCHD$, as a set of N scheduling assignments:

$$SCHD = \{ b_i : i = 1, 2, ..., N \},$$

and a *partial schedule*, p, as a subset of such a complete schedule:

$$p \subseteq \{ b_i : i = 1, 2, ..., N \}.$$

The task of scheduling is then to optimize $SCHD$ subject to the following constraints:

1. Data Flow Dependency Constraints

   For all integers i and j such that $i \neq j$, $1 \leq i \leq N$ and $1 \leq j \leq N$, if there is an edge from n_i to n_j in the CDFG, then:

$$s_i + d(n_i) \leq s_j$$

where $d(n_i)$ is the delay of the operation associated with node n_i (measured in number of control steps).

2. Maximum Value Constraint

All nodes must end in or before control step G:

$$\underset{i=1}{\overset{N}{Max}} (s_i + d(n_i) - 1) \leq G$$

Let C_{Schd} denote the objective cost function for the scheduling problem. Assuming that scheduling is performed prior to allocation, C_{Schd} cannot calculate the actual circuit area. Instead, we define C_{Schd} as the ***minimum*** area [47] required by the complete schedule, $SCHD$. This includes the minimum areas of functional units (*operators* from now on), registers and interconnects.

The minimum area of operators is determined by the maximum numbers of *concurrent* nodes with the same operation types, since all concurrent nodes must be allocated to separate operators to avoid resource contention conflicts. Similarly, the minimum areas of registers and interconnects are determined by the maximum numbers of concurrent edges and data transfers, respectively. Basically, two nodes, edges, or data transfers are concurrent if their *lifetimes* overlap. Ignoring pipelined operators for now, the lifetime of a node n_i starts from control step s_i and ends in control step $s_i + d(n_i) - 1$. On the other hand, the lifetime of an edge e produced by n_i and accessed by nodes in the set $dests(e)$ starts from control step $s_i + d(n_i) - 1$ and ends in the control step

$$(\underset{n_j \in dests(e)}{Max} s_j + d(n_j) - 1) - 1.$$

That is, the lifetime of e ends in the control step just before it is last accessed (see Section 2.3.4). Finally, for each edge e produced by a node n_i, a data transfer is required in control step $s_i + d(n_i) - 1$ to store the data value corresponding to e in a register, and a data transfer is required from control step s_j to $s_j + d(n_j) - 1$ for each node n_j which accesses e.

To facilitate the calculation of C_{Schd} and other costs, we define a *tally* data structure to keep track of the minimum hardware requirements due to overlapping lifetimes. Conceptually, the tally is a two dimensional array which records concurrent CDFG elements by operation types and control step numbers. For each node n_i with operation type $op(n_i)$ and a lifetime from control step i to j, we add n_i to the sets $tally(op(n_i),t)$ for $t = i, i+1, \ldots, j$. Moreover, for each edge e with a lifetime from i to j, we add e to the sets $tally(R,t)$ for $t = i, i+1, \ldots, j$, where R designates the operation type for registers. Finally, for each data transfer due to e with a lifetime from i to j, we add e to the sets $tally(T,t)$ for $t = i, i+1, \ldots, j$, where T designates the operation type for interconnects. This counts the number of *distinct* edges that require data transfers in each control step, in effect assuming that the interconnect cost is dominated by the number of busses required.

Given the above, we define the objective cost function, C_{Schd}, as:

$$C_{Schd} = \sum_{op \in OPS} c(op) \times \underset{t=1}{\overset{G}{Max}} \, |tally(op,t)|$$

where OPS is the set of all operation types used in the tally data structure, including the types R and T.

Given a partial schedule p, we say that a node n_i is *scheduled* to control step cs if $b_i = (n_i, cs) \in p$, that n_i is *unscheduled* if $b_i \notin p$, and that n_i is *de-scheduled* if b_i is removed from the set p. Consequently, the *GENERATE* function in SE corresponds to a greedy scheduling algorithm, and the *SELECT* function corresponds to a probabilistic de-scheduling algorithm. In the following sections, we define the cost functions *PRIORITY*, *INCR* and *GLOBAL* which completely characterize our SE-based scheduling algorithm.

5.3. The PRIORITY Function

The function *PRIORITY* calculates the priority of each unscheduled node to determine the sequence in which they are processed by *GENERATE*. Following the practice of solving the most constrained tasks first, we assign the highest priority to the node with the minimum number of scheduling candidates. This results in a freedom-based priority scheduling similar to that used by MAHA [46]. However, unlike MAHA in which nodes with equal freedoms are scheduled in an arbitrary order, we use the freedoms of neighboring nodes to break such ties.

The *freedom* of an unscheduled CDFG node n_i, denoted by $f(n_i)$, is defined as the number of control steps in which a node n_i may be scheduled without violating any data flow dependency constraints. Given a partially scheduled CDFG, we define an *as soon as possible* schedule for n_i, denoted by $asap(n_i)$, as the smallest *legal* value for s_i, and define an *as late as possible* schedule for n_i, denoted by $alap(n_i)$, as the largest legal value for s_i. By definition, if n_i is already scheduled, then $asap(n_i) = alap(n_i) = s_i$. The freedom for n_i is calculated as:

$$f(n_i) = alap(n_i) - asap(n_i) + 1 \; .$$

To take into account the freedoms of neighboring nodes, we define the *starting freedom* of a CDFG edge e, denoted by $f_{start}(e)$, as the number of control steps in which the lifetime of e may start. This is calculated as:

$$f_{start}(e) = alapStart(e) - asapStart(e) + 1$$

where $alapStart(e)$ and $asapStart(e)$ are the as late as possible and as soon as possible start times of e, respectively. Let $src(e)$ denote the CDFG node which produces the edge e. Then $alapStart(e)$ and $asapStart(e)$ are defined as:

$$alapStart(e) = alap(src(e)) + d(src(e)) - 1$$

and

$$asapStart(e) = asap(src(e)) + d(src(e)) - 1.$$

Moreover, we define the *ending freedom* of an edge e, denoted by $f_{end}(e)$, as the number of control steps in which the lifetime of e may end. This is calculated as:

$$f_{end}(e) = alapEnd(e) - asapEnd(e) + 1$$

where $alapEnd(e)$ and $asapEnd(e)$ are the as late as possible and as soon as possible end times of e, respectively. Let $dests(e)$ denote the set of nodes which access the edge e. Then $alapEnd(e)$ and $asapEnd(e)$ are defined as:

$$alapEnd(e) = \underset{n \in dests(e)}{Max} (alap(n) + d(n) - 2)$$

and

$$asapEnd(e) = \underset{n \in dests(e)}{Max} (asap(n) + d(n) - 2).$$

Given the above, we define *PRIORITY* as:

$$PRIORITY(n_i) = -\,[\,f(n_i) + w \times (\sum_{e \in inputs(n_i)} f_{start}(e) + \sum_{e \in outputs(n_i)} f_{end}(e)\,)\,]$$

where $inputs(n_i)$ is the set of edges accessed by n_i, $outputs(n_i)$ is the set of edges produced by n_i, and $w \geq 0$ is a control parameter which determines the relative importance between node and edge freedoms. For example, if w is small, then $PRIORITY(n_i)$ depends primarily on $f(n_i)$. On the other hand, if w is large, then $PRIORITY(n_i)$ depends primarily on the cumulative starting (ending) freedoms of the edges accessed (produced) by n_i.

Initially, we assigned a small value for w so as to give priority to node freedoms. However, extensive experiments showed that a large value of w produces the best scheduling performance. This was puzzling until we recognized that the cumulative edge freedoms actually measure the freedoms of the predecessor and certain successor nodes. The fact that a large value of w results in better scheduling performance is evidence that the *neighborhood freedoms* of a node are more important than the actual node freedom in determining the sequence in which nodes are scheduled. This is confirmed by the dominance of the *incremental opportunity cost* component (see next section) as demonstrated by experimental results in Chapter 9.

5.4. The INCR Function

The function *INCR* determines the control step to which an unscheduled node is assigned. Given an unscheduled node n_i and a scheduling candidate cs ($asap(n_i) \leq cs \leq alap(n_i)$), $INCR(n_i, cs)$ estimates the increase in C_{Schd} if n_i is scheduled to cs. This is a sum of four component costs:

$$INCR(n_i, cs) = I_{op}(n_i, cs) + I_R(n_i, cs) + I_T(n_i, cs) + I_P(n_i, cs)$$

where I_{OP} is the incremental operator cost, I_R is the incremental register cost, I_T is the incremental interconnect cost, and I_P is the incremental opportunity cost due to the decreased freedoms in other unscheduled nodes.

5.4.1. Incremental Operator Cost

The cost function $I_{OP}(n_i, cs)$ measures the increased operator area when n_i is scheduled to control step cs. Basically, if scheduling n_i to control step cs requires an additional operator of type $op(n_i)$, then I_{OP} is $c(op(n_i))$ (i.e., area cost for one operator of type $op(n_i)$), otherwise I_{OP} is 0. If the tally data structure is up-to-date (see below), then an additional operator is required for the candidate schedule if and only if there is already a maximum number of operations of type $op(n_i)$ in any of the sets $tally(op(n_i), t)$ for $t = cs, cs+1, \ldots, cs+d(n_i)-1$. Hence we define I_{OP} as:

1. $I_{OP}(n_i, cs) = c(op(n_i))$ if

$$\underset{t=1}{\overset{G}{Max}} |tally(op, t)| = \underset{t=cs}{\overset{cs+d(n_i)-1}{Max}} |tally(op, t)|$$

2. Otherwise, $I_{OP}(n_i, cs) = 0$

At the start of the scheduling algorithm, $tally(op, t)$ is set to $\varnothing$ for all operation types op and all control steps t. Subsequently, whenever a node n_i is scheduled, we add n_i to the sets $tally(op(n_i),t)$ for $t = s_i, s_i+1, \ldots, s_i+d(n_i)-1$ to reflect the scheduled lifetime of n_i. On the other hand, whenever n_i is de-scheduled, we delete n_i from the sets $tally(op(n_i),t)$ for $t = s_i, s_i+1, \ldots, s_i+d(n_i)-1$ to keep the *tally* data structure up-to-date.

5.4.2. Incremental Register Cost

The cost function $I_R(n_i, cs)$ measures the increased register area due to scheduling n_i to control step cs. When n_i is scheduled to cs, all CDFG edges accessed by n_i (i.e., $inputs(n_i)$) will have lifetimes which last until at least control step $cs + d(n_i) - 2$, and all edges produced by n_i (i.e., $outputs(n_i)$) will have lifetimes which start from control step $cs + d(n_i) - 1$. If these lifetimes result in more concurrent edges than the current maximum, then additional registers are required. Unfortunately, calculating the exact register cost is expensive because it requires excessive updates of the tally data structure. Instead, we calculate the increased register area for each edge assuming it is the only edge accessed or produced by n_i, and then approximate I_R as either the sum of area increases due to edges in $inputs(n_i)$, or the sum of area increases due to edges in $outputs(n_i)$, whichever is larger:

$$I_R(n_i, cs) = Max\left(\sum_{e \in inputs(n_i)} I_{iR}(e, cs+d(n_i)-2), \sum_{e \in outputs(n_i)} I_{oR}(e, cs+d(n_i)-1) \right)$$

where $I_{iR}(e,t)$ is the increased register area if e is *live* at least until control step t, and $I_{oR}(e,t)$ is the increased register area if e is live from control step t. These are calculated in basically the same way as incremental operator costs are calculated:

1. $I_{iR}(e,t) = c(R)$ if

$$\underset{i=1}{\overset{G}{Max}} |tally(R,i)| = \underset{i=L}{\overset{t}{Max}} |tally(R,i) - \{e\}|$$

where L is a lower bound control step defined as $L = Min(alapStart(e), t)$,

2. Otherwise, $I_{iR}(e,t) = 0$.

and,

1. $I_{oR}(e,t) = c(R)$ if

$$\underset{i=1}{\overset{G}{Max}} \; |tally(R,i)| = \underset{i=t}{\overset{U}{Max}} \; |tally(R,i) - \{e\}|$$

where U is an upper bound control step defined as $U = Max(asapEnd(e)\ ,\ t)$,

2. Otherwise, $I_{oR}(e,t) = 0$.

At the start of *GENERATE*, we update $tally(R,t)$ as follows. First, $tally(R,t)$ is set to $\varnothing$ for all control steps t. Then for each edge e in the CDFG, we add e to the sets $tally(R,t)$ for $t = alapStart(e)\ ,\ alapStart(e){+}1\ ,...,\ asapEnd(e)$. This initializes the minimum number of concurrent edges even if the complete lifetimes for some of these edges are not fixed yet (i.e., edges e with $alapStart(e) \leq asapEnd(e)$). Subsequently, whenever a node n_i is scheduled, we update $asapStart(e)$ and $alapStart(e)$ for all edges $e \in outputs(n_i)$, and propagate these changes to the *asap* times for all descendent nodes of n_i, and the *asapStart* times for all edges produced by these nodes. Similarly, whenever n_i is scheduled, we update $asapEnd(e)$ and $alapEnd(e)$ for all edges $e \in inputs(n_i)$, and propagate these changes to the *alap* times for all ancestor nodes of n_i, and the *alapEnd* times for all edges accessed by these nodes. Then for each edge e with a different $alapStart(e)$ or $asapEnd(e)$, we add e to the sets $tally(R,t)$ for $t = alapStart(e)\ ,\ alapStart(e){+}1\ ,...,\ asapEnd(e)$ to reflect the new (partial) lifetime of e. (Note that $tally(R,t)$ is a set, hence adding e to $tally(R,t)$ when e is already in the set does not change $tally(R,t)$). However, we do not incrementally update $tally(R,t)$ after de-scheduling a node in *SELECT* since it can get very complicated. Instead, we invalidate the entire $tally(R,t)$ data structure whenever a node is de-scheduled, and then reconstruct it at the start of the next *GENERATE* step.

5.4.3. Incremental Interconnect Cost

The cost function $I_T(n_i,cs)$ measures the increased interconnect area due to scheduling n_i to control step cs. When n_i is scheduled to cs, each edge accessed by n_i requires a data transfer for control steps $t = cs$, $cs+1$,..., $cs+d(n_i)-1$, and each edge produced by n_i requires a data transfer for control step $cs+d(n_i)-1$. If these data transfer requirements lead to more concurrent data transfers than the current maximum, then additional interconnects are required, resulting in an increase in interconnect costs.

We define $I_T(n_i,cs)$ as:

$$I_T(n_i,cs) = Max(I_{ioT}(n_i,cs+d(n_i)-1)\ ,\ \underset{t\,=\,cs}{\overset{cs+d(n_i)-2}{Max}}\ I_{iT}(n_i,t))$$

where $I_{ioT}(n_i,t)$ is the increased interconnect cost when all edges accessed and produced by n_i require data transfers in control step t, and $I_{iT}(n_i,t)$ is the increased interconnect cost when all edges accessed by n_i require data transfers in control step t. These are calculated as follows:

$$I_{ioT}(n_i,t) = c(T) \times Max(0\ ,\ |inputs(n_i) \cup outputs(n_i) \cup tally(T,t)| - MC(T))$$

and

$$I_{iT}(n_i,t) = c(T) \times Max(0\ ,\ |inputs(n_i) \cup tally(T,t)| - MC(T))$$

where $MC(T)$ is the maximum number of concurrent distinct data transfers:

$$MC(T) = \underset{t\,=\,1}{\overset{G}{Max}}\ |tally(T,t)|.$$

At the start of *GENERATE*, we first set $tally(T,t)$ to $\varnothing$ for all control steps t. Then for each scheduled node n_i, we add the edges $inputs(n_i)$ to the sets $tally(T,t)$ for $t = s_i$, s_i+1 ,..., $s_i+d(n_i)-1$, and add the edges $outputs(n_i)$ to the set $tally(T,s_i+d(n_i)-1)$. Subsequently, whenever a node n_i is scheduled, we add the edges $inputs(n_i)$ and $outputs(n_i)$ to $tally(T,t)$ in exactly the same way. However, we do not incrementally update $tally(T,t)$ whenever a node is de-scheduled, but invalidate the entire *tally*(T, t) data structure and then reconstruct it at the start of the next *GENERATE* step.

5.4.4. Incremental Opportunity Cost

The cost function $I_P(n_i, cs)$ measures the increased risk of additional operator costs due to scheduling n_i to control step cs. When n_i is scheduled to cs, the freedoms for other unscheduled nodes may be reduced due to data flow dependency. This increases the risk that additional area increases may be required when these nodes are subsequently scheduled. Unfortunately, calculating the exact cost of this increased risk is expensive since it involves propagating the effects of each candidate schedule throughout the entire CDFG, and a detailed analysis of these effects. Instead, we only consider the first order effects, namely the probable increases in operator areas for the immediate predecessor and successor nodes of n_i.

Let $pred(n_i)$ denote the set of all predecessor nodes of n_i, and let $succ(n_i)$ denote the set of all successor nodes of n_i. Then whenever a node n_i is scheduled to cs, each predecessor node $n_p \in pred(n_i)$ must be scheduled no later than $cs - d(n_p)$, and each successor node $n_s \in succ(n_i)$ must be scheduled no earlier than $cs + d(n_i)$. Consequently, we define $I_P(n_i, cs)$ as:

$$I_P(n_i, cs) = \sum_{p \in pred(n_i)} C_p(p\ , asap(p)\ , cs - d(p)) + \sum_{s \in succ(n_i)} C_p(s\ , cs + d(n_i)\ , alap(s))$$

where $C_p(n, t_1, t_2)$ calculates the probable increase in operator costs for a node n if its scheduling freedom is reduced to between control steps t_1 and t_2 inclusive. This is calculated as:

$$C_p(n, t_1, t_2) = c(op(n)) \times \left(\frac{1}{t_2 - t_1 + 1} - \frac{1}{f(n)}\right)$$

This opportunity cost penalizes those candidate schedules which unduly constrain subsequent scheduling of neighboring nodes, and basically implements an one-step look-ahead to the greedy scheduling algorithm. In particular, this opportunity cost biases the scheduling algorithm so that nodes without inputs tend to be scheduled to the earlier control steps, and nodes without outputs tend to be scheduled to the later control steps. The importance of this cost component to scheduling performance will be demonstrated by experimental results in Chapter 9.

5.5. The GLOBAL Function

The function *GLOBAL* determines the probability that *SELECT* will de-schedule a CDFG node. For each node n_i, $GLOBAL(n_i)$ calculates a pro-rated cost for n_i based on its contribution to the overall cost, C_{Schd}, of the scheduled CDFG. This is a sum of three component costs:

$$Global(n_i) = G_{OP}(n_i) + G_R(n_i) + G_T(n_i)$$

where G_{OP} is the pro-rated operator cost, G_R is the pro-rated register cost, and G_T is the pro-rated interconnect cost.

We pro-rate the operator cost for n_i as follows:

$$G_{OP}(n_i) = c(op(n_i)) \times \frac{\underset{t = s_i}{\overset{s_i+d(n_i)-1}{Max}} \; |tally(op(n_i),t)|}{\underset{t=1}{\overset{G}{Max}} \; |tally(op(n_i),t)|}$$

That is, if n_i is live in a control step with the maximum number of concurrent operations of the same type (i.e., $op(n_i)$), then $G_{OP}(n_i) = c(op(n_i))$; otherwise $G_{OP}(n_i)$ is assigned a fraction of $c(op(n_i))$ depending on the maximum number of concurrent operations of type $op(n_i)$ during the lifetime of n_i.

We pro-rate the register cost for each CDFG edge, e, in a similar way:

$$G_E(e) = c(R) \times \frac{\underset{t = asapStart(e)}{\overset{alapEnd(e)}{Max}} \; |tally(R,t)|}{\underset{t=1}{\overset{G}{Max}} \; |tally(R,t)|}$$

and then attribute this cost equally to the node which produces e and the set of nodes $dests_L(e)$ which last access e:

$$dests_L(e) = \{\ n_j : e \in inputs(n_j) \text{ and } alapEnd(e) = alap(n_j) + d(n_j) - 2\ \}.$$

Consequently, we define $G_R(n_i)$ as:

$$G_R(n_i) = \sum_{e \in IO(n_i)} \frac{G_E(e)}{1 + |dests_L(e)|}$$

where $IO(n_i)$ is the set of edges which are either produced or last accessed by the node n_i:

$$IO(n_i) = outputs(n_i) \cup \{\ e : n_i \in dests_L(e)\ \}$$

Finally, we pro-rate the interconnect cost for each data transfer in control step t due to edge e as:

$$G_{DT}(e,t) = c(T) \times \frac{|tally(T,t)|}{\underset{t=1}{\overset{G}{Max}}\ |tally(T,t)|}$$

and then attribute this cost to the interconnect cost of the node accessing or producing e in control step t. For each node n_i, the pro-rated interconnect cost is then:

$$G_T(n_i) = Max(\ \underset{t=s_i}{\overset{s_i+d(n_i)-2}{Max}}\ G_{iT}(n_i,t)\ ,\ G_{ioT}(n_i,s_i+d(n_i)-1)\ )$$

where $G_{iT}(n_i,t)$ is the sum of the pro-rated interconnect costs for all edges accessed by n_i, and $G_{ioT}(n_i,t)$ is the sum of the pro-rated interconnect costs for all edges accessed and produced by n_i. These are calculated as:

$$G_{iT}(n_i,t) = \sum_{e \in inputs(n_i)} G_{DT}(e,t)$$

and

$$G_{ioT}(n_i,t) = \sum_{e \in inputs(n_i) \cup outputs(n_i)} G_{DT}(e,t).$$

5.6. Summary

In summary, we have described a SE-based scheduling algorithm in this chapter by first formulating scheduling as an optimum assignment problem, and then defining the cost functions *PRIORITY*, *INCR* and *GLOBAL*. The function *PRIORITY* determines the order in which unscheduled nodes are selected for scheduling. This implements a freedom-based heuristic similar to MAHA's. However, whereas MAHA gives priority to nodes with small scheduling freedoms, our implementation gives priority to nodes with small *neighborhood freedoms* as determined by the cumulative edge freedoms for all edges produced and accessed by each node. The cost function *INCR* estimates the increase in C_{Schd} due to each scheduling candidate. This includes an opportunity cost, which serves as a one-step look-ahead for the greedy scheduling algorithm in *GENERATE*. Finally, the cost function *GLOBAL* calculates the pro-rated hardware cost for each node based on its contribution to the overall cost, C_{Schd}. This is defined as the sum of the pro-rated operator, register and interconnect costs.

Chapter 6. Basic Allocation Algorithm

In this chapter, we will first review previous allocation approaches, and then describe our application of SE to the allocation task. To do so, we will first formulate the basic allocation task as an optimum assignment problem, and then describe our SE-based allocation algorithm by defining the cost functions *PRIORITY*, *INCR* and *GLOBAL*.

6.1. Previous Allocation Approaches

Existing allocation algorithms can be classified into two groups: namely the iterative/constructive group and the global group [37]. An iterative/constructive allocation algorithm successively assigns hardware to operations, data values and data transfers until a complete data path is constructed. Different systems in this group differ in their heuristics for selecting the next operation, data value or data transfer to be allocated, and in the cost functions which determine the best allocation candidates. Splicer [43], DAA [25], EMUCS [15], Elf [11], and MABAL [27] are examples of iterative/constructive allocation algorithms. Among these synthesis systems, Splicer allows a branch-and-bound search for better designs, while DAA explores the design space by applying redesign rules in its expert knowledge base.

On the other hand, global allocation algorithms apply graph theoretic or mathematical programming techniques to the allocation problem or its subtasks. Facet [57] and HAL [47] formulate the allocation of operations, data values and data transfers as three separate clique-partitioning problems (see Section 3.1). However, clique-partitioning is an NP-hard problem, so heuristics are used to solve the resulting problems. LYRA [16] formulates the allocation of operations and data values as two *bipartite weighted matching* problems, and solves these using the Hungarian method [44]. The allocation of data transfers is then solved using greedy heuristics. By dividing the allocation task into three separate problems, these global techniques do not permit tradeoffs between functional units, registers and interconnects in the data path. Hafer [13] formulates the allocation task as a mixed integer linear programming problem, and applies branch and bound to solve a small example.

In applying SE to the allocation task, we implement an iterative/constructive allocation algorithm which differs from previous allocation algorithms in that it probabilistically explores the design space using a "rip-up and re-allocate" approach.

6.2. Allocation as an Optimum Assignment Problem

Given a CDFG containing N *elements* (i.e., nodes *and* edges), denoted by e_i ($i = 1, 2, \ldots, N$), and a *scheduled lifetime* for each CDFG element, we formulate the task of allocation as an optimum assignment problem as follows. Let a_i ($i = 1, 2, \ldots, N$) denote a variable whose value specifies the cell instance (simply *cell* from here on) on which the CDFG element e_i is realized, and let I denote a finite set of cells sufficient to implement each CDFG element on a separate cell (i.e., the maximally parallel allocation). We define the binding of *inst* to a_i, denoted by $b_i = (a_i, inst)$ ($i = 1, 2, \ldots, N$ and $inst \in I$), as *allocating* the CDFG element e_i to the cell *inst*. Then we define a *complete allocation*, denoted by *ALLOC*, as a set of N such allocation assignments:

$$ALLOC = \{ b_i : i = 1, 2, \ldots, N \},$$

and a *partial allocation*, p, as a subset of such a complete allocation:

$$p \subseteq \{ b_i : i = 1, 2, \ldots, N \}.$$

The task of allocation is then to optimize *ALLOC* subject to the following constraint:

Resource Contention Constraint

> For all integers i and j such that $i \neq j$, $1 \leq i \leq N$ and $1 \leq j \leq N$, if e_i and e_j are concurrent (i.e., have overlapping lifetimes), then $a_i \neq a_j$.

Let C_{Alloc} denote the objective cost function for the allocation problem. Assuming that scheduling is performed before allocation, C_{Alloc} can calculate the circuit area required by $ALLOC$. This includes the area for all cells allocated to the CDFG elements, and the area for the interconnects required to implement all data transfers between these cells.

The cell area required depends on the numbers and types of *distinct* cells in the complete allocation $ALLOC$. Let $I(op)$ denote the set of cells allocated to CDFG nodes (or CDFG edges if $op = R$) with type op:

$$I(op) = \{ a_i : op(e_i) = op \ , i = 1 \ , 2 \ ,..., N \ \}$$

where $op(e_i)$ is the operator type of e_i if e_i is a CDFG node, and $op(e_i) = R$ if e_i is a CDFG edge. Then the total area for cells required by $ALLOC$ is:

$$\sum_{op \in OPS} c(op) \times | I(op) |$$

where OPS is the set of all operator types in the CDFG, including the type R for CDFG edges.

On the other hand, the interconnect areas required by $ALLOC$ depends on the numbers and types of multiplexers required to implement all of the required data transfers. We define a data transfer from the n'th output (n being a positive integer) of a CDFG element e_i to the m'th input of another CDFG element e_j as the quadruple $dt = (e_i, n, e_j, m)$. Ignoring commutative operations for now (see Section 7.8), the interconnect required to implement this data transfer is a circuit connection from the n'th output signal of the cell a_i to the m'th input signal of a_j, i.e., $int(dt) = (a_i, n, a_j, m)$. Let INT denote the set of all distinct interconnect required by $ALLOC$:

$$INT = \{ int(dt) : dt \in DTS \ \}$$

where DTS denotes the set of all data transfers in the CDFG. Assuming the one-level-multiplexer scheme of interconnects, an M-to-1 multiplexer is required for each input signal with a *fan-in* of $M > 1$. The fan-in of, say, the m'th input signal of a cell *inst* is the number of circuit connections ending at the input signal. We denote this as $fi(inst, m)$, formally defined by:

$$fi(inst, m) = | \{ \ int(dt) : int(dt) \in INT \text{ and } dt = (a_i, n, inst, m) \ \} |.$$

Consequently, the total interconnect area for $ALLOC$ is:

$$\sum_{op \in OPS} \ \sum_{inst \in I(op)} \ \sum_{m} c_M(fi(inst, m))$$

where $c_M(n)$ is the area for an n-to-1 multiplexer.

Given the above, the objective cost function C_{Alloc} is defined as:

$$C_{Alloc} = \sum_{op \in OPS} \left(c(op) \times |I(op)| + \sum_{inst \in I(op)} \ \sum_{m} c_M(fi(inst, m)) \right).$$

As stated in Section 2.3.4, we assume that every n-input multiplexer (or bus) is constructed from $n-1$ 2-input multiplexers, and the area of a wire is negligible (i.e., $c_M(1) = 0$):

$$c_M(n) = \begin{cases} 0 & \mathit{If} \ n \le 1 \\ (n-1) \times c_M(2) & \mathit{Otherwise} \end{cases}$$

Hence we measure the quality of an allocation by the required numbers of registers, operators, and *equivalent* 2-input multiplexers.

Given a partial allocation p, we say that a CDFG element e_i is *allocated* to cell *inst* if $b_i = (a_i, inst) \in p$, that e_i is *unallocated* if $b_i \notin p$, and that e_i is *de-allocated* if b_i is removed from the set p. Consequently, the *GENERATE* function in an SE-based allocation algorithm corresponds to a greedy allocation algorithm, and the *SELECT* function corresponds to a probabilistic de-allocation algorithm. In the following sections, we define the cost functions *PRIORITY*, *INCR* and *GLOBAL* which completely characterize our SE-based allocation algorithm.

6.3. The PRIORITY Function

The function *PRIORITY* calculates the priority of each unallocated CDFG element to determine the sequence in which they are processed by *GENERATE*. This is defined as a weighted sum of four measures:

$$PRIORITY(e_i) = w_1 \times cost(e_i) + w_2 \times span(e_i) + w_3 \times alloc(e_i) + w_4 \times conn(e_i)$$

where $cost(e_i) = c(op(e_i))$ is the area cost of a new cell for e_i, $span(e_i)$ is the number of control steps in the lifetime of e_i, $alloc(e_i)$ is the total number of *allocated neighbors* of e_i (i.e., allocated CDFG elements which have data transfers to and from e_i), $conn(e_i)$ is the total number of data transfers to and from e_i, and w_1, w_2, w_3 and w_4 are decreasing weights.

The motivation for this formulation lies in the way the greedy allocation algorithm calculates the *incremental* interconnect costs. Basically, the greedy allocation algorithm allocates each CDFG element, e_i, to a *new* cell unless the *incremental* interconnect costs of allocating e_i to an *existing* cell is less than $gf \times c(op(e_i))$, where gf is a control parameter that trades off interconnect and cell costs. The cost function *INCR* is defined such that it only considers data transfers between e_i and its allocated neighbors when calculating these interconnect costs (see the next section), hence the algorithm tends to under-estimate the interconnect costs of e_i if it is allocated earlier, when fewer of its neighbors have been allocated. Consequently, the greedy algorithm tends to allocate earlier CDFG elements to existing cells, and later CDFG elements to new cells, resulting in poor tradeoffs between interconnect costs and cell costs. To alleviate this short-sightedness in the greedy allocation algorithm, we try to create new cells early so that they may be allocated to subsequent CDFG elements, and we try to create many allocated neighbors as soon as possible so that more accurate interconnect costs may be used when allocating subsequent CDFG elements.

Given the above, we now present an intuitive argument for the above formulation of *PRIORITY*. First, CDFG elements with high cell costs should be allocated to a small number of cells in spite of high interconnect costs, hence we may safely underestimate interconnect costs when allocating such CDFG elements. Since interconnect costs tend to be under-estimated in the earlier allocations, we give priority to CDFG elements with high $cost(e_i)$. Second, CDFG elements with long lifespans are more likely to be allocated to new cells because they are more likely to be concurrent with other CDFG elements already allocated to existing cells. By giving priority to elements with high $span(e_i)$, we try to create new cells as early as possible. Third, CDFG elements with many allocated neighbors tend to have higher incremental interconnect costs for allocating to existing cells, and are therefore more likely to be allocated to new cells. By giving priority to CDFG elements with high $alloc(e_i)$, we try to create new cells early for the sake of subsequent allocations. Finally, CDFG elements with many data transfers tend to have more neighbors, hence their allocations have wider impact in terms of the number of allocated neighbors that subsequent CDFG elements will have. By giving priority to CDFG elements with high $conn(e_i)$, we try to increase the numbers of allocated neighbors as much as possible to facilitate subsequent allocations.

The weights w_1, w_2, w_3 and w_4 are assigned decreasing values to reflect the relative importance of their corresponding factors. The exact values of these weights are set by experimentation.

6.4. The INCR Function

The function *INCR* determines which cell is allocated to each CDFG element. Given an unallocated element e_i and a partial allocation p, the allocation candidates consist of all existing cells of type $op(e_i)$ which satisfy all resource contention constraints with e_i, plus a new cell of the same type. In this context, an existing cell *inst* satisfies all resource contention constraints of e_i if and only if none of the CDFG elements already allocated to *inst* is concurrent (i.e., has overlapping lifetime) with e_i. Since a new cell has not been allocated to any CDFG element, it is always a legal allocation candidate for e_i.

For each candidate allocation, say ca, of e_i, $INCR(e_i, ca)$ estimates the increase in C_{Alloc} when e_i is allocated to ca. If ca is an existing cell, then *INCR* calculates the incremental interconnect costs required to implement all data transfers between e_i and its allocated neighbors:

$$INCR(e_i, ca) = \sum_{dt \in DTS(e_i)} I_T(e_i, ca, dt)$$

where $DTS(e_i)$ is the set of all data transfers which originate from or are destined for e_i, and $I_T(e_i, ca, dt)$ is the additional interconnect area required to implement the data transfer dt *assuming* $a_i = ca$ (i.e., that e_i is allocated to ca). For example, if dt is a data transfer which originates from e_i, then we have $dt = (e_i, n, e_j, m)$ for some n, j and m, and the interconnect required is $int(dt) = (ca, n, a_j, m)$. On the other hand, if dt is destined for e_i, then we have $dt = (e_j, n, ca, m)$, and the interconnect required is $int(dt) = (a_j, n, ca, m)$. In either case, we define I_T as follows:

1. If e_j is not allocated, then $I_T(e_i, ca, dt) = 0$;
2. If e_j is allocated, and the interconnect $int(dt)$ already exists in the circuit, then $I_T(e_i, ca, dt) = 0$;

3. If e_j is allocated, and $int(dt)$ does not exist in the circuit, then I_T is calculated as the cost for increasing the fan-in of the destination signal of $int(dt)$ by 1:

   3.1. If $int(dt) = (ca, n, a_j, m)$, then

$$I_T(e_i, ca, dt) = c_M(1 + fi(a_j, m)) - c_M(fi(a_j, m))$$

   3.2 Otherwise (i.e., $int(dt) = (a_j, n, ca, m)$),

$$I_T(e_i, ca, dt) = c_M(1 + fi(ca, m)) - c_M(fi(ca, m))$$

That is, $INCR(e_i, ca)$ only considers the cost of adding the *missing* interconnects between ca and the cells allocated to the neighboring CDFG elements of e_i if ca is an existing cell.

However, if ca is a new cell, then $INCR$ is simply a weighted cell cost of ca:

$$INCR(e_i, ca) = gf \times c(op(e_i))$$

where $gf > 0$ (for *generosity factor*, borrowed from MABAL [27]) is a control parameter. Basically, a new cell will be allocated to e_i if and only if the incremental interconnect costs for all existing candidate cells of e_i exceed $gf \times c(op(e_i))$. Thus the control parameter gf determines how the greedy allocation algorithm trades off interconnect and cell costs. If gf is very small (e.g., 0.5), then the algorithm will allocate new cells unless the incremental interconnect cost for allocating to an existing cell is nearly 0. On the other hand, if gf is very large (e.g., 10.0), then the algorithm will only allocate new cells when none of the existing cells is a legal allocation candidate. The default value of gf is set to 1.1, which seemed to produce the best results in our experiments.

6.5. The GLOBAL Function

The function *GLOBAL* determines the probability that *SELECT* will de-allocate a CDFG element. For each element e_i, $GLOBAL(e_i)$ calculates a pro-rated cost for e_i based on its contribution to the overall cost, C_{Alloc}, of the allocation CDFG. This is a sum of two component costs:

$$GLOBAL(e_i) = G_{OP}(e_i) + G_T(e_i)$$

where G_{OP} is the pro-rated cell cost, and G_T is the pro-rated interconnect cost.

To facilitate the calculation of these costs, we define a *justification list* for each cell and each interconnect in the circuit. Basically, whenever a cell, ca, is allocated to a CDFG element, e_j, we add e_j to the justification list of ca, denoted by $J(ca)$; whenever e_j is de-allocated, we remove e_j from the justification list $J(a_j)$. Similarly, whenever an interconnect, *int*, is used to implement a data transfer, dt, we add dt to the justification list $J(int)$; whenever dt is no longer required due to a CDFG element being de-allocated, we remove dt from the list $J(int(dt))$. In a complete allocation, these justification lists fully specify the CDFG elements and data transfers which make use of each cell and interconnect in the allocated circuit.

Given the above, we may define the pro-rated cell cost, G_{OP}, as:

$$G_{OP}(e_i) = \frac{c(op(e_i))}{|J(a_i)|}$$

That is, $G_{OP}(e_i) = c(op(e_i))$ if the cell a_i is only used by the CDFG element e_i. Otherwise, the cell cost of a_i is equally divided among all CDFG elements in $J(a_i)$ (i.e., CDFG elements to which a_i is allocated).

The pro-rated interconnect cost, G_T, is more difficult to define directly. Instead, we define a pro-rated cost for each data transfer, and then equally divide this cost between the two CDFG elements involved in the data transfer. Specifically, for each data transfer $dt = (e_i, n, e_j, m)$, we define the pro-rated data transfer cost, denoted by

$G_{DT}(dt)$, as:

$$G_{DT}(dt) = \frac{c_M(fi(a_j,m))}{fi(a_j,m)} \times \frac{1}{|\, J(int(dt))\, |} .$$

That is, the interconnect cost of $int(dt)$ is calculated as the cost per input for the multiplexer required at the m'th input of the cell a_j (i.e., $\frac{c_M(fi(a_j,m))}{fi(a_j,m)}$). This interconnect cost is then divided equally among all data transfers which make use of $int(dt)$ (i.e., $J(int(dt))$). Finally, we distribute each pro-rated data transfer cost, $G_{DT}(dt)$, equally between the two CDFG elements involved in the data transfer (i.e., e_i and e_j).

Consequently, we define the pro-rated interconnect cost, G_T, as:

$$G_T(e_i) = \sum_{dt \,\in\, DTS(e_i)} \frac{G_{DT}(dt)}{2} .$$

That is, $G_T(e_i)$ is the sum of half portions of the pro-rated costs for all data transfers which originate from or are destined for e_i.

6.6. Summary

In summary, we have described a SE-based allocation algorithm in this chapter by first formulating allocation as an optimum assignment problem, and then defining the cost functions *PRIORITY*, *INCR* and *GLOBAL*. The function *PRIORITY* determines the order in which CDFG elements are allocated. It tries to alleviate the short-sightedness in the greedy allocation algorithm by giving priority to elements which are more likely to be allocated to new cells. The function *INCR* estimates the increase in C_{Alloc} due to each proposed allocation. If the allocation candidate is an existing cell, *INCR* calculates the extra interconnect area required; if the allocation candidate is a new cell, then *INCR* calculates a weighted cell cost. This implements user control of the tradeoffs between interconnect and cell costs in the greedy allocation algorithm. Finally, the function *GLOBAL* calculates the pro-rated hardware cost of each CDFG element based on its contribution to the overall cost, C_{Alloc}. This is defined as a sum of the pro-rated cell and interconnect costs.

Chapter 7. Refined Synthesis Algorithms

In this chapter, we will describe a number of extensions to the scheduling and allocation algorithms presented in Chapters 5 and 6. Our objective is not to describe the implementation details of each extension, since most of these extensions have been implemented by previous synthesis systems in one way or another. Instead, our goal is to show that such extensions are easily implemented by changing the cost functions in SE-based synthesis algorithms. Extensions for the following are considered:

- Hardware Constraints
- Local Timing Constraints
- Case Constructs
- Loops
- Area/Time Tradeoffs
- Operation Chaining
- Equivalent Allocation Constraints
- Operators with Permutable Inputs
- Interconnect Optimization
- Catastrophic Rip-up in Allocation
- Structural Pipelining
- Algorithmic Pipelining

Extensions for allocation with incomplete schedules and scheduling with partial allocation will be deferred to Chapter 10.

7.1. Hardware Constraints

Maximum constraints on the numbers of specific cells (e.g., "at most 3 adders" or "at most 10 registers") are handled by adding a penalty cost for constraint violations to the cost functions *INCR* and *GLOBAL*. This biases the scheduling and allocation algorithms against violating maximum hardware constraints. On the other hand, minimum constraints (e.g., "at least 7 interconnects") are treated as user specified circuit initialization, and provides a look-ahead to the *INCR* cost function in both the scheduling and allocation algorithms.

7.2. Local Timing Constraints

Local timing constraints may be specified between any two (groups of) CDFG nodes using control edges in the CDFG (see Section 2.3.5.1). Recall that a minimum timing constraint of $T_{\min}$ control steps from n_i to n_j is represented by a control edge from n_i to n_j with a positive delay of $T_{\min}$. On the other hand, a maximum timing constraint of $T_{\max}$ control steps from n_i to n_j is represented by a control edge from n_i to n_j with a negative delay of $-T_{\max}$ (see Fig. 2.7).

We extend the scheduling algorithm to handle minimum timing constraints by modifying the definitions of $asap(n_i)$ and $alap(n_i)$ (for an unscheduled node n_i) to consider control edges with positive delays. However, maximum timing constraints are handled by adding a penalty cost for constraint violations to the functions *INCR* and *GLOBAL* to bias the algorithm against violating such constraints.

7.3. Case Constructs

Nodes and edges in different conditional branches of a case construct are mutually exclusive (see Section 2.3.5.1), and will never be live simultaneously even though they may have overlapping lifetimes. To accommodate case constructs, we extend the notion of *concurrency* to including mutual exclusion: two CDFG elements are concurrent if and only if they are *not* mutually exclusive *and* their lifetimes overlap. The scheduling and allocation algorithms are modified accordingly to reflect this extended definition of concurrency.

7.4. Loops

For CDFG containing loops, we schedule each loop separately, starting with the inner most loop if there are nested loops. Each loop is then treated as a special multiple control-step operation which requires exclusive use of the data path. A simple reservation scheme is implemented in the scheduling algorithm which marks all control steps in the lifetime of a loop operation as unavailable for (the lifetime of) any other node. Also, the minimum hardware requirements within each scheduled loop are treated as minimum hardware constraints for the entire CDFG, thus providing look-aheads to subsequent scheduling steps.

7.5. Area/Time Tradeoffs

We have extended the scheduling algorithm to explore area/time tradeoffs using objective functions of the form

$$A \times (T - T_{\min})^N \tag{5.1}$$

or

$$A + N \times (T - T_{\min}) \tag{5.2}$$

where A is the area cost of the (partial) schedule, T is the total number of control steps required by the (partial) schedule, $T_{\min}$ is the (user specified) minimum number control steps in the final schedule, and N is a user specified constant. For each node n_i, we define $alap_{\min}(n_i)$ as the "as late as possible" schedule if the entire CDFG must be scheduled in $T_{\min}$ control steps. We then extend the cost function $GLOBAL(n_i)$ to multiply the pro-rated area costs by

$$Max(1\ ,\ (s_i - alap_{\min}(n_i))^N)$$

if the objective function is of the form (5.1), or increase the pro-rated area costs by

$$N \times Max(1\ ,\ (s_i - alap_{\min}(n_i)))$$

if the objective function is of the form (5.2). The cost function *INCR* is extended in a similar way, except we dynamically adjust $alap_{\min}$ to track changes in the critical path lengths as CDFG nodes are scheduled and de-scheduled.

7.6. Operation Chaining

Operation chaining refers to scheduling a node to the same control step as its predecessor and/or successor CDFG nodes. This is feasible if the cumulative delay for the chained operations is less than the duration of the data path phase (i.e., $\overline{\phi}$) of the clock. Operation chaining is especially useful when scheduling time critical sequences of fast and inexpensive operations such as logic operations (e.g., AND, OR and NOT). To implement operation chaining, we extend the delay model of operations to include *setup* and *hold* times. For operations with single control step delays, the setup and hold times are equal, and specify operation delays in nanoseconds. For pipelined operations, the setup time specifies the delay of the first pipe-stage, and the hold time specifies the delay in the last pipe-stage of the operation. For non-pipelined operations with multiple control step delays, the setup time is the duration of the data path phase ($\overline{\phi}$), and the hold time is the operation delay modulo the period of the clock, ϕ. This prevents chaining a non-pipelined, multiple control step operation with its predecessor because all inputs to the operation must have been stored in registers in order to remain stable for the duration of that operation.

Given the setup and hold times for all CDFG nodes, we extend the calculation of $asap(n_i)$ and $alap(n_i)$ to consider subcycle timing in order to determine if n_i may be chained with its predecessor and successor nodes. For example, let n_i be the last predecessor node for n_j, and let n_j in turn be the first successor node for n_i. Given that n_i ends at h nanoseconds after the start of control step cs, if $d(n_j) = 1$ (or if n_j is implemented by a pipelined cell) and the setup time for n_j is less than or equal to $\overline{\phi} - h$, then n_j may be chained with n_i and $asap(n_j)$ is set to cs, otherwise $asap(n_j)$ is set to $cs+1$. On the other hand, given that n_j starts at k nanoseconds before the end of the $\overline{\phi}$ phase in control step cs, if $d(n_i) = 1$ (or if n_i is implemented by a pipelined cell) and the hold time for n_i is less than or equal to $\overline{\phi} - k$, then n_i may be chained with n_j and $alap(n_i)$ is set to $cs - d(n_i) + 1$, otherwise $alap(n_i)$ is set to $cs - d(n_i)$. We do not permit chaining of a non-pipelined node, n_i, with its predecessor nodes if it has a multi-cycle delay (i.e., $d(n_i) > 1$), since the inputs of n_i cannot be held stable past the $\overline{\phi}$ phase of control step s_i if n_i is chained with its predecessors. Finally, we modify the algorithms to handle CDFG edges with empty lifetimes (i.e., edges whose $asapStart < alapEnd$) due to chaining. Such edges do not require storage, but are implemented as direct data transfers between operators.

7.7. Equivalent Allocation Constraints

Equivalent allocation constraints specify that two or more CDFG elements must be allocated to the same cells. We use these constraints to ensure that all edges representing the same logical data (e.g., in loops and case constructs) are allocated to the same register. These constraints are also useful as a mechanism for direct user control of the allocation algorithm. Basically, we replace each set of equivalent CDFG elements by a "super" CDFG element which merges all inputs and outputs of the original CDFG elements. These "super" CDFG elements are then allocated in exactly the same way as regular CDFG elements.

7.8. Operators with Permutable Inputs

Certain operators have permutable inputs in the sense that the data to certain inputs of these operators may be permuted arbitrarily. For example, a 2-input simple adder may have 2 permutable inputs and hence 2 ways to connect its input data to its input ports, while a 3-input *AND*-gate may have 3 permutable inputs and hence 6 different ways to connect its input data to its 3 input ports. To minimize interconnect costs, we extend the allocation algorithm to consider all permutations of input connections and assume the minimum cost permutations when calculating the cost function *INCR* in the allocation algorithm.

7.9. Interconnect Optimization

We have interfaced the allocation algorithm to an interconnect optimization algorithm [33] which reduces the total interconnect area by transforming the one-level-multiplexer style of interconnects to a multi-level style of interconnects (i.e., one in which two or more multiplexers or busses may be cascaded in series). This interconnect optimization is invoked after greedy allocation, so that the optimized interconnect areas are used in determining the best allocations. However, since this interconnect optimization algorithm is quite slow, invoking it in every SE iteration would excessively slow down the allocation speed. Consequently, we invoke interconnect optimization only if the circuit area prior to optimization is within a percentage, say p, of the minimum circuit area found so far. The percentage p may be specified by users, or dynamically updated as the *maximum percentage improvement that interconnect optim-*

ization has ever achieved on the current CDFG. This sacrifices some accuracy for computational speed, but still provides an override mechanism for user control.

7.10. Catastrophic Rip-Up in Allocation

To further perturb SE-based allocation, we have implemented a *catastrophic rip-up* step which is randomly invoked with a low (parameterized) probability at the start of *SELECT*. When invoked, this step first evaluates each cell, ca, in the circuit on the basis of its overhead cost per cell area:

$$\frac{\sum_{e \in J(ca)} Global(e, ca)}{c(type(ca))} > 1$$

where $J(ca)$ is the justification list of ca (i.e., the set of all CDFG elements allocated to ca). Basically, this measures the total cost of ca, including the interconnect and penalty costs incurred by all CDFG elements allocated to ca, as a multiple of the cell area of ca (i.e., $c(type(ca))$). The catastrophic rip-up step then probabilistically removes cells with high overhead costs by de-allocating all CDFG elements allocated to these cells.

7.11. Structural Pipelining

In structural pipelining, two or more CDFG nodes may execute concurrently on different stages in the same pipelined cell. We define the *latency* of a node n_i, denoted by $l(n_i)$, as a positive integer which specifies the number of control steps during which n_i requires *exclusive* use of an operator. If n_i is to be implemented on a pipelined operator, then $l(n_i)$ is set to the latency of the pipeline. On the other hand, if n_i is to be implemented on a non-pipelined operator, then $l(n_i)$ is set to the delay, $d(n_i)$.

Given the above, we redefine the lifetime of each node n_i in terms of $l(n_i)$

instead of $d(n_i)$ (i.e., from s_i to $s_i + l(n_i) - 1$ inclusive), in the sense that n_i only requires the *exclusive* use of an operator for as many control steps as its latency, $l(n_i)$. This also redefines the lifetimes of CDFG edges and data transfers, since each node n_i only accesses its inputs until control step $s_i + l(n_i) - 1$, but generates its outputs in control step $s_i + d(n_i) - 1$. The cost functions *INCR* and *GLOBAL* for the scheduling algorithm are modified accordingly to reflect the new lifetimes for nodes, edges and data transfers.

7.12. Algorithmic Pipelining

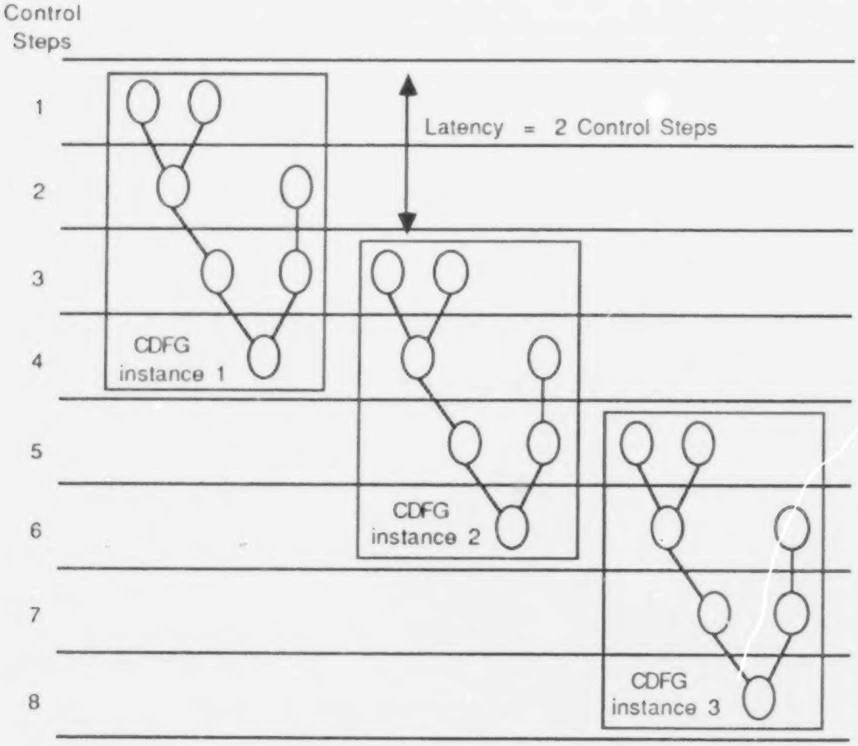

Control Steps
1
2
3
4
5
6
7
8
Latency = 2 Control Steps
CDFG instance 1
CDFG instance 2
CDFG instance 3

Figure 7.1 Algorithmic Pipelining: Consecutive Instances of a LOOP Overlap in Time

In algorithmic pipelining, successive instances of a CDFG loop body execute concurrently in the same data path in order to increase throughput at the expense (if any) of a slightly larger data path [47]. We define the *latency*, denoted by L, of an algorithmic pipeline as a positive integer which specifies the number of control steps between consecutive instantiations of a CDFG loop body. Basically, CDFG elements which are live in control steps $i + k \times L$ ($k = 0, 1, 2, \cdots$) will be concurrent due to the overlapping execution of different instances of the loop (see Fig. 7.1). Consequently, we treat the control steps $i + k \times L$ ($k = 0, 1, 2, \cdots$) as *aliases* of the control step i, and define a new access function, $|tally'(op, t)|$, for the tally data structure to include elements which are live in all aliased control steps of a control step t:

$$|tally'(op, t)| = \sum_k |tally(op, t + k \times L)|$$

Chapter 8. Design Examples for SE-based Synthesis

We have implemented the SE-based scheduling and allocation algorithms presented in chapters 5 to 7 in about 6000 lines of Common Lisp and Flavors code on a SUN 4 running SUN Common Lisp 4.0. In this chapter, we present experimental results of our SE-based scheduling and allocation algorithms on a number of design examples from the literature. These results demonstrate that, compared to other synthesis systems, our SE-based synthesis algorithms generate comparable designs quickly, and generate better designs when given longer run times.

The CPU times reported for our results are measured on a SUN 4 running SunOS 4.1 with 24 MB of RAM and 72 MB of swap space. We have tried some of these examples on a SUN 4 running SunOS 4.1.1 with 32 MB of RAM and 125 MB of swap space, and observed a speed up by as much as a factor of 2.

8.1. Differential Equation Example from HAL

The CDFG for this example (Fig. 8.1) is taken from [47]. We schedule and allocate this CDFG under two sets of assumptions as in [47]. In the first case, we assume a maximum global timing constraint of 4 control steps, and that additions, subtractions and comparisons are performed by adders, subtractors and comparators, respectively, whose delays are half the duration of $\overline{\phi}$ (i.e., up to two of such operations may be chained), and multiplications are performed by multipliers whose delays are the duration of $\overline{\phi}$ (i.e., no chaining allowed). Also, we assume that constants (i.e., dX, a, and 3) are hard-wired to power and ground. Under these assumptions, it takes 8 scheduling iterations (at an average of 0.35 CPU second per iteration) to obtain a schedule requiring a minimum of 1 adder, 1 subtractor, 2 multipliers, 1 comparator, 5 registers and 8 interconnects. However, it takes 32 scheduling iterations (at 0.25 second per iteration) to find a schedule requiring the same amount of operators and registers but

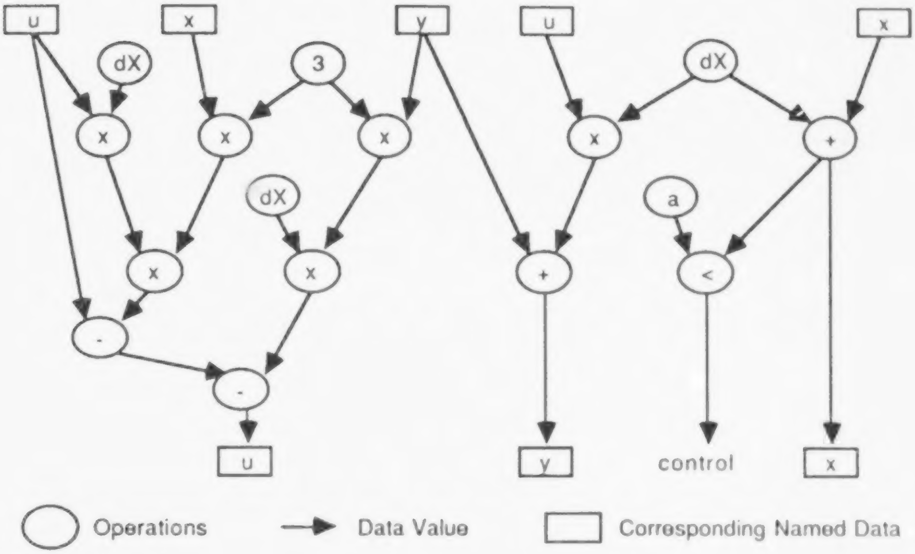
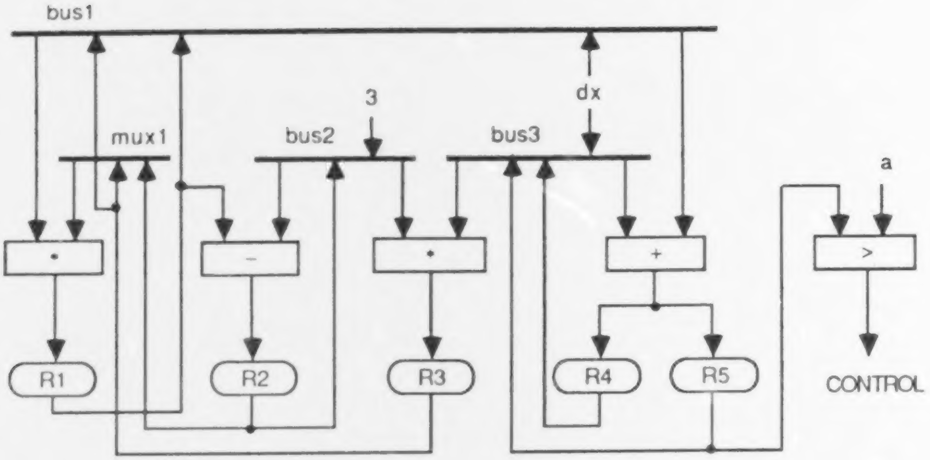


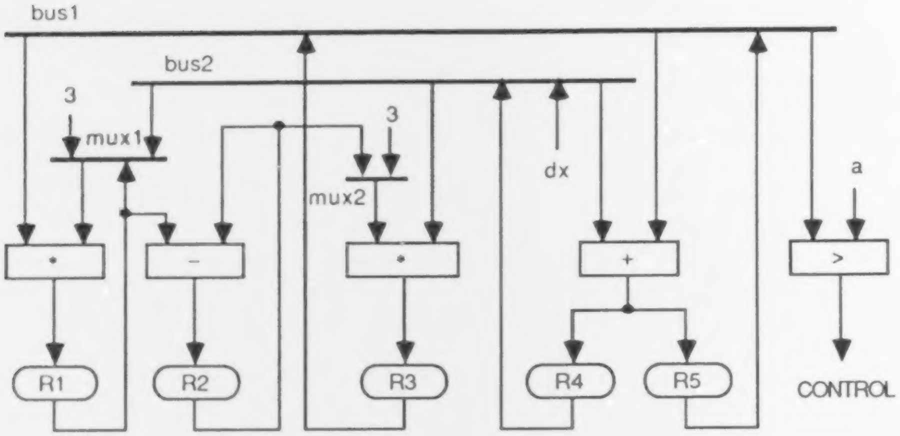
Figure 8.1 CDFG for the Differential Equation Example

only 7 interconnects. Given this schedule, it takes 43 allocation iterations (at an average of 0.32 second per iteration) to obtain a design containing the minimum number of operators and registers, and with less interconnects than the best design reported to date [47]. Fig. 8.2 compares our circuit with HAL's circuit for this example. Table 8.1 compares our design against those reported by Splicer [43], CATREE [10] and HAL [47] for the same example.

In the second case, we assume a maximum global timing constraint of 8 control steps, and assume that additions, subtractions and comparisons are performed by ALU's in one control step without operation chaining, and multiplications are performed by (structurally) pipelined multipliers in two control steps. Under these assumptions, it takes 5 scheduling iterations (at 0.6 second per iteration) to obtain the best schedule requiring 1 ALU, 1 multiplier, 5 registers and 5 interconnects. Given this schedule, it takes only 2 allocation iterations (at 1.5 second per iteration) to find a


bus1
mux1
bus2
3
dx
bus3
a
*
–
*
+
>
R1
R2
R3
R4
R5
CONTROL

(a) HAL's Circuit


bus1
bus2
3
mux1
3
mux2
dx
a
*
–
*
+
>
R1
R2
R3
R4
R5
CONTROL

(b) SSE's Circuit

Figure 8.2 Circuits for Differential Equation Example (4 Control Steps)

design which is slightly better than the best design reported to date [47] (Fig. 8.3(a)). However, it takes 466 allocation iterations (at an average of 0.64 second per iteration) to find the best circuit we have ever obtained for this example. This circuit is shown

	FU	R	Mx	Mi	ME
Splicer	5	6	5	12	7
CATREE	5	6	7	12	5
HAL	5	5	4	10	6
SSE	5	5	4	9	5

FU: Functional Units R: Registers
Mx: Muxes/Busses Mi: Mux/Bus Inputs
ME: Equivalent 2-to-1 Muxes (Mi - Mx)

Table 8.1 Design Costs for Differential Equation Example (4 Control Steps)

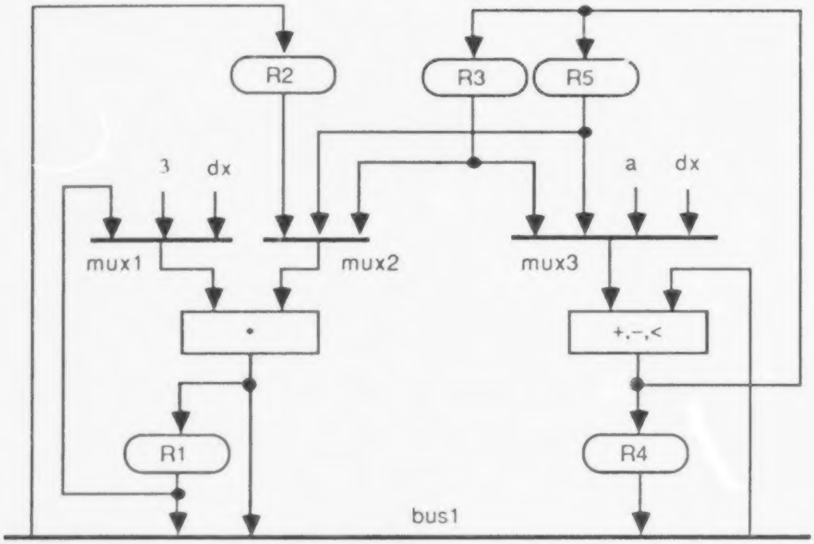
in Fig. 8.3(b). Table 8.2 compares our design against those generated by Splicer [43] and HAL [47].

	FU	R	Mx	Mi	ME
Splicer	3 *	7 **	6	16	10
HAL	2	5	4	13	9
SSE	2	5	5	12	7

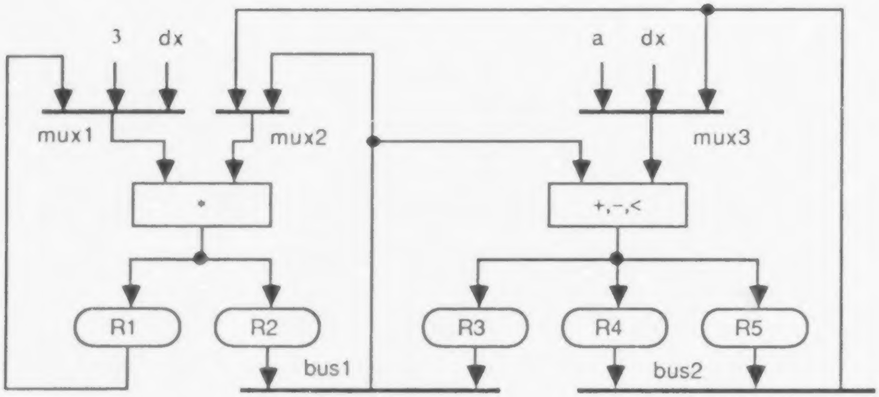
FU: Functional Units R: Registers
Mx: Muxes/Busses Mi: Mux/Bus Inputs
ME: Equivalent 2-to-1 Muxes
* 1 Multiplier, 1 Comparator and 1 ALU
** Constants Stored in Registers

Table 8.2 Design Costs for Differential Equation Example (8 Control Steps)

Note that the Splicer design assumes that the comparison operation is implemented on


R2
R3
R5
3
dx
a
dx
mux1
mux2
mux3
*
+,−,<
R1
R4
bus1

(a) HAL's Circuit


3
dx
a
dx
mux1
mux2
mux3
*
+,−,<
R1
R2
R3
R4
R5
bus1
bus2

(b) SSE's Circuit

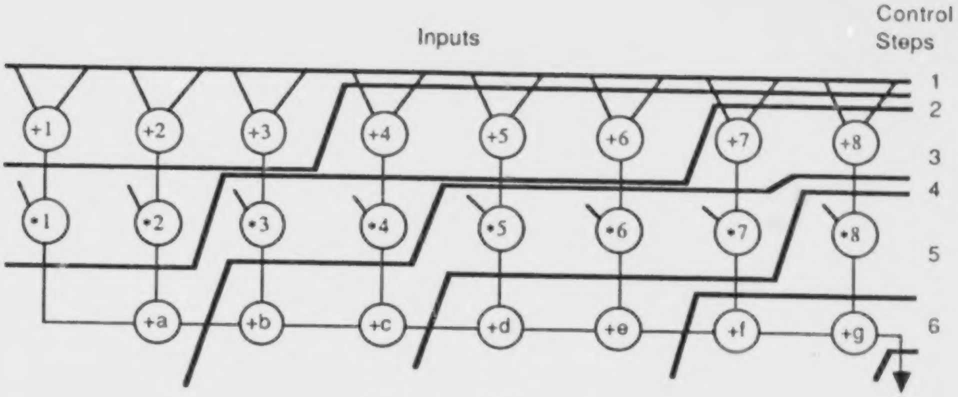
Figure 8.3 Circuits for Differential Equation Example (8 Control Steps)

a comparator and that all constants are stored in registers.

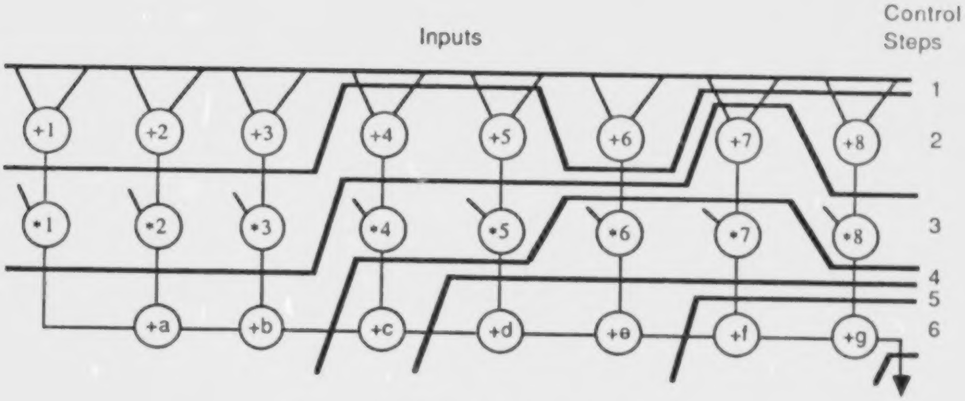
8.2. Pipelined Examples from Sehwa

Two examples are taken from [45] to demonstrate scheduling with algorithmic pipelining, operation chaining, and conditional branches. The first example is a 16-point digital FIR filter with a global timing constraint of 6 control steps and an algorithmic pipelining latency of 3 control steps. We assume additions are performed by adders in half the duration of $\bar{\phi}$ (i.e., up to 2 additions may be chained), and multiplications are performed by multipliers in a full duration of $\bar{\phi}$. For this example, it takes just 1 scheduling iteration (in 1.13 CPU second) to find a schedule comparable to that obtained by Sehwa's exhaustive scheduling method [45] (i.e., requiring 5 adders, 3 multipliers and 24 registers). However, it takes 17 scheduling iterations (at 0.44 second per iteration) to find a schedule requiring one less register (i.e., 5 adders, 3 multipliers and 23 registers). Fig. 8.4(a) shows the schedule produced by Sehwa's exhaustive method, and Fig. 8.4(b) shows the schedule produced by 17 iterations of SE-based scheduling. Note that due to algorithmic pipelining, the minimum number of registers required by these schedules is determined by the total number of edges entering control steps 1 and 4, plus the number of edges leaving control step 6. Table 8.3 compares our schedule with those generated by both the backward feasible scheduling method and the exhaustive scheduling methods in Sehwa [45], and by HAL [47].

The second example from Sehwa contains mutually exclusive operations in an algorithmically pipelined CDFG. Nodes and edges from different stages in the algorithmic pipeline are counted as requiring separate hardware resources if their pipestages overlap in time, even if they belong to mutually exclusive branches of a case construct. For this example, it takes 31 scheduling iterations (at 0.65 second per iteration) to find a schedule which is identical to that obtained by Sehwa's exhaustive scheduling [45], which is also obtained by HAL [47]. This schedule is shown in Fig. 8.5.


Inputs
Control Steps
1
2
3
4
5
6
+1
+2
+3
+4
+5
+6
+7
+8
•1
•2
•3
•4
•5
•6
•7
•8
+a
+b
+c
+d
+e
+f
+g

(a) Sehwa's Schedule


Inputs
Control Steps
1
2
3
4
5
6
+1
+2
+3
+4
+5
+6
+7
+8
•1
•2
•3
•4
•5
•6
•7
•8
+a
+b
+c
+d
+e
+f
+g

(b) SSE's Schedule

—— Control Step Boundary

Figure 8.4 Schedules for the FIR Filter Example

	Control Steps	+	x	R	T
Sehwa (Feasible)	7	6	3	31	21
Sehwa (Exhaustive)	6	5	3	24	20
HAL	6	5	3	-	-
SSE	6	5	3	23	20

+: Adders

x: Multipliers

R: Registers

T: Interconnects

Table 8.3 Schedule Costs for the FIR Filter Example

8.3. Fifth Order Elliptic Wave Filter (EWF) Example

The CDFG for the fifth order Elliptic Wave Filter (EWF) example is taken from [47] and is shown in Fig. 8.6. This example was chosen as a benchmark for the 1988 High Level Synthesis Workshop [2]. We assume that additions are performed on adders in one control step without operation chaining, multiplications are performed on multipliers in two control steps, and all filter coefficients are stored in a small ROM which is directly connected to one of the inputs of the multipliers. Under these assumptions, we schedule this CDFG using different global timing constraints and with both pipelined and non-pipelined multipliers. The results are summarized in Table 8.4. Note that the CPU time for SE-based scheduling deteriorates as the scheduling freedom increases. This is to be expected since, with increasing scheduling freedoms, there are more scheduling candidates to evaluate (i.e., scheduling steps are slower) and a larger design space to explore (i.e., more SE iterations are required).

After scheduling, we allocate each of these schedules assuming permutable adder

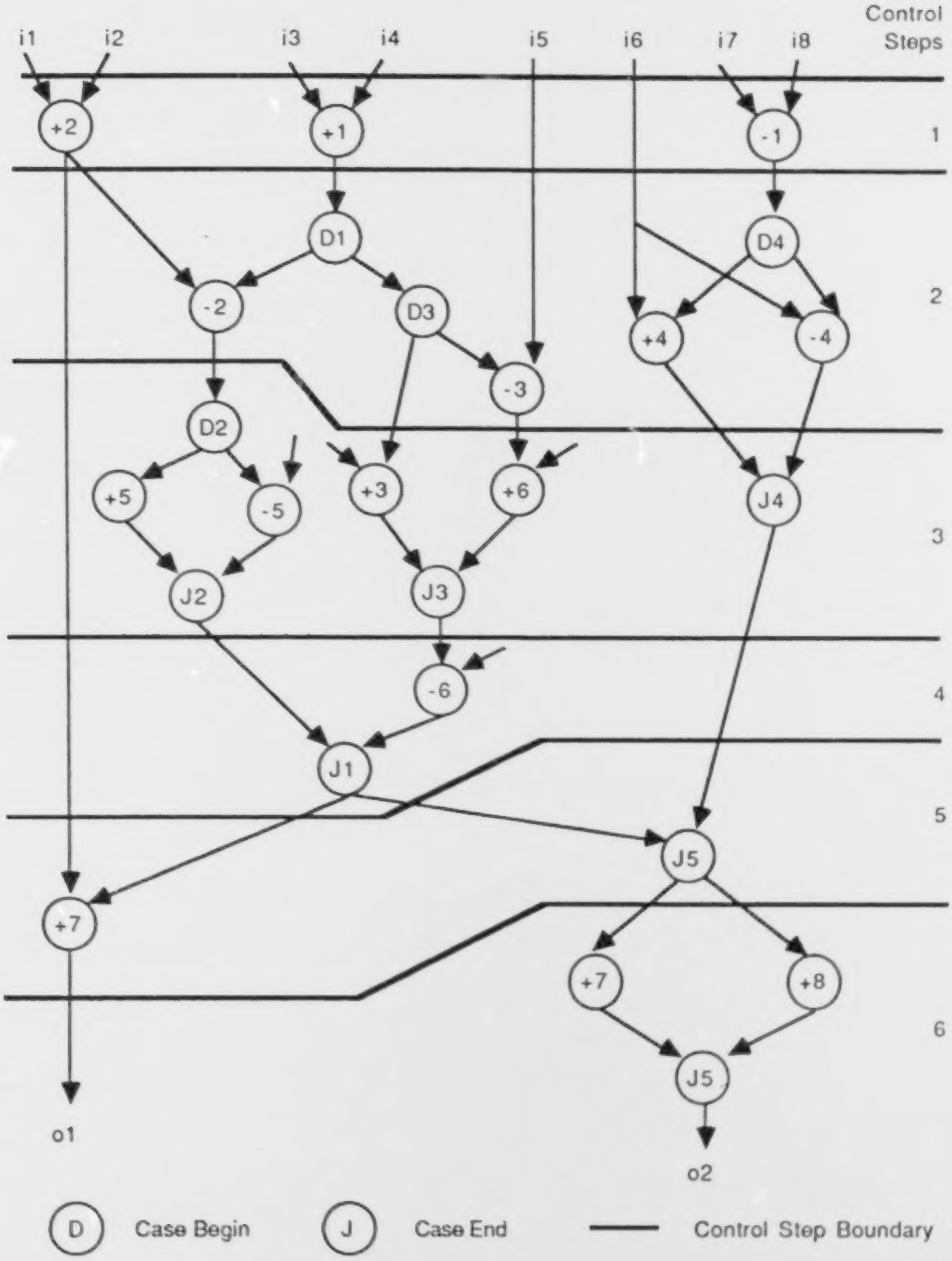

i1
i2
i3
i4
i5
i6
i7
i8
Control Steps
+2
+1
-1
1
D1
D4
-2
D3
+4
-4
2
-3
D2
+5
-5
+3
+6
J4
3
J2
J3
-6
4
J1
5
J5
+7
+7
+8
6
J5
o1
o2
D
Case Begin
J
Case End
Control Step Boundary

Figure 8.5 Scheduled Example with Algorithmic Pipeline and CASE Constructs

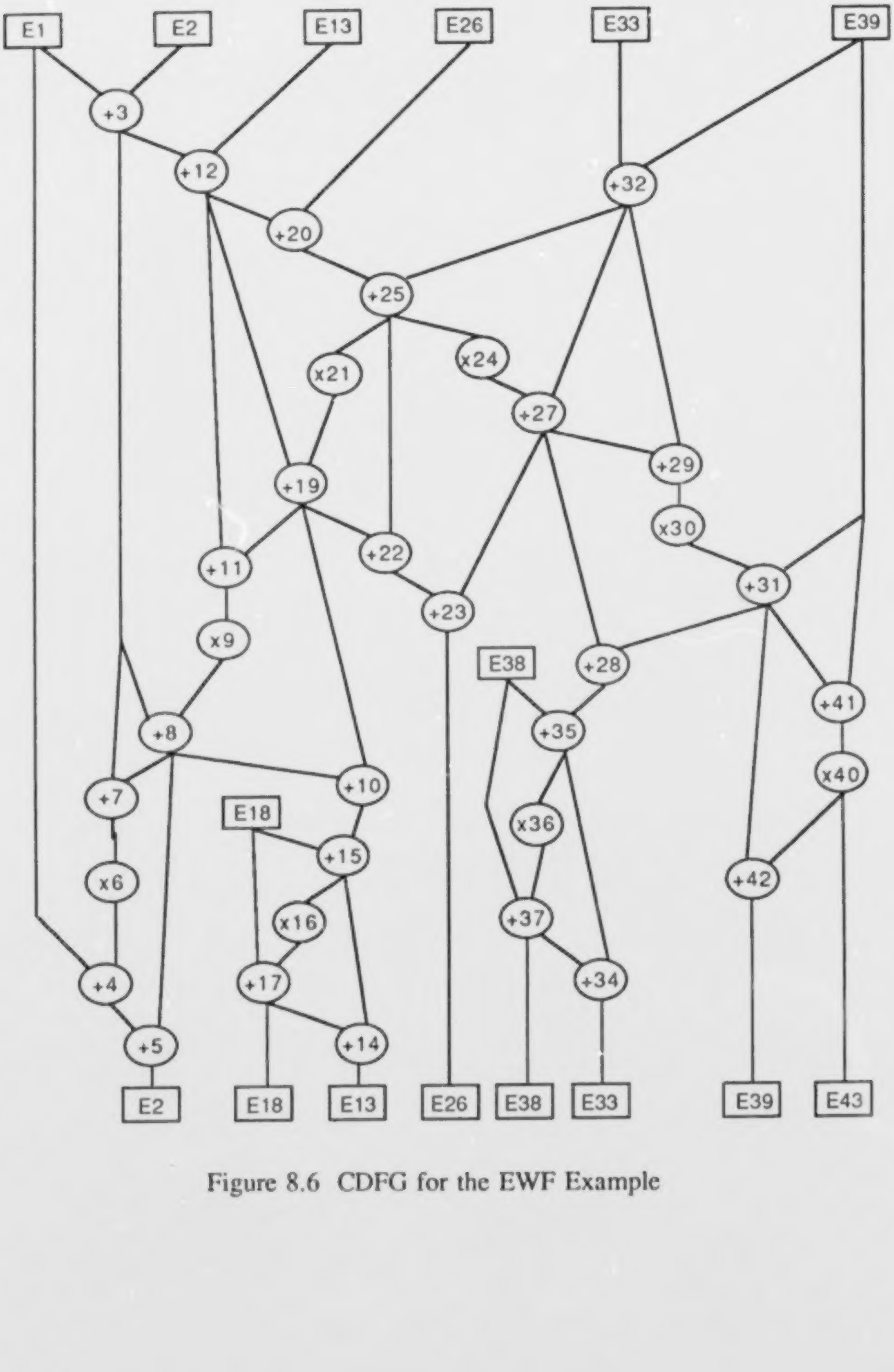

E1
E2
E13
E26
E33
E39
+3
+12
+32
+20
+25
x21
x24
+27
+29
+19
x30
+22
+11
+31
+23
x9
E38
+28
+41
+8
+35
x40
+7
+10
E18
x36
+15
x6
+42
x16
+37
+4
+17
+34
+5
+14
E2
E18
E13
E26
E38
E33
E39
E43

Figure 8.6 CDFG for the EWF Example

Control Steps	Minimum Cost	Iterations	CPU Time (s)
17	3+, 2xP, 10R, 10T	3	2.39
18	3+, 1xP, 10R, 8T	37	24.05
19	2+, 1xP, 10R, 7T	20	14.40

(a) With Pipelined Multipliers

Control Steps	Minimum Cost	Iterations	CPU Time (s)
17	3+, 3x, 11R, 11T	2	1.62
18	2+, 2x, 10R, 9T	38	23.94
19	2+, 2x, 10R, 8T	37	24.79
20	2+, 2x, 10R, 8T	17	13.26
21	2+, 1x, 10R, 8T	123	105.78

(b) With Non-Pipelined Multipliers

+: Adders
x: Multipliers
xP: Pipelined Multipliers
R: Registers
T: Interconnects

Table 8.4 Scheduling Performance for the EWF Example

inputs and with interconnect optimization enabled. The best results are compared with those reported in [27, 48] and summarized in Table 8.5. In all cases, our designs use the same numbers of operators but fewer registers and less interconnects than those produced by other systems.

Table 8.6 shows the schedule and allocation for the case of (a maximum global timing constraint of) 19 control steps and using pipelined multipliers. In this table, each CDFG edge is named according to the node which produces the edge (e.g., edge $E5$ is generated by node +5 and $E6$ by node $x6$), except for edges $E1$, $E2$, $E13$,

System	Control Steps	FU	R	Mx	Mi	ME	Time (s)
SSE	17	2xP, 3+	11	12	31	19	1511
HAL	17	2xP, 3+	12	--	31	--	--
CATREE	17	2xP, 3+	12	--	38	--	--
SSE	19	2x, 2+	10	11	31	20	1791
HAL	19	2x, 2+	12	--	29	--	--
EMUCS	19	2x, 2+	12	12	34	22	--
SSE	19	1xP, 2+	11	9	25	16	2096
HAL	19	1xP, 2+	12	6	26	20	--
SSE	21	1x, 2+	11	8	24	16	2870
HAL	21	1x, 2+	12	--	31	--	--
SPLICER	21	1x, 2+	--	9	43	34	--
MABAL	21	2x, 2+	11	13	43	30	--

Table 8.5 Design Costs for the EWF Example

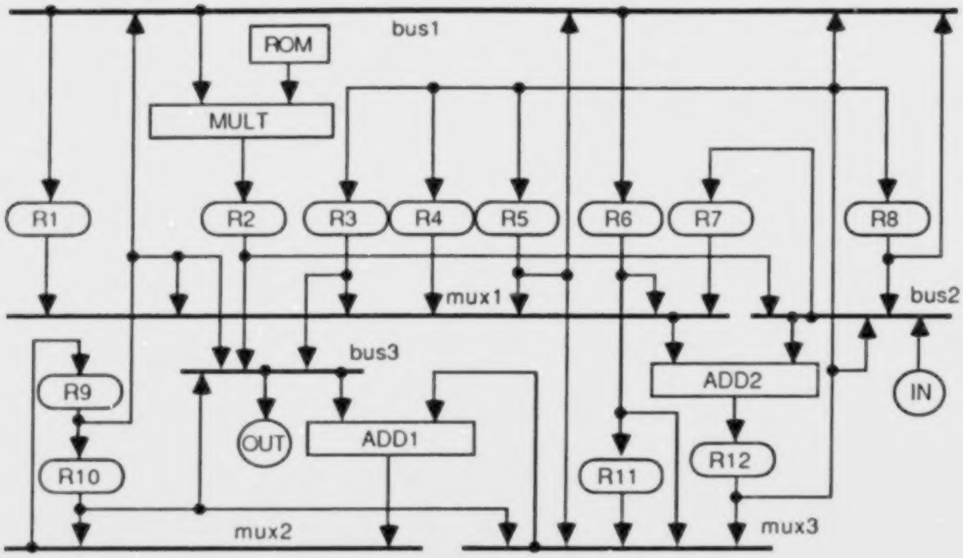
$E26$, $E33$, $E39$, $E18$, $E38$, and $E43$ which represent equivalent edges (drawn as square boxes in Fig. 8.6). Fig. 8.7 compares this circuit with HAL's circuit (which is still the best one besides ours) for the same set of constraints. Notice that our design (Fig. 8.7(b)) has a regular architecture: most of the registers are dedicated to storing values from single (and at most, two) sources; all multiplexers and busses are small (with only four or fewer inputs); and a simple two-level-multiplexer style of interconnects is used. Moreover, it is interesting to note that this design happens to use a

	ADD1	ADD2	MULT	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11
IN					E18	E13	E33		E39	E26	E2		E38	E1
1		+3			E18	E13	E33		E39	E26	E3		E38	E1
2		+12			E18		E33		E39	E26	E3	E12	E38	E1
3	+20	+32			E18	E20	E32		E39		E3	E12	E38	E1
4	+25			E25	E18		E32		E39		E3	E12	E38	E1
5			x24	E25	E18		E32		E39		E3	E12	E38	E1
6			x21	E25	E18		E32	E24	E39		E3	E12	E38	E1
7		+27		E25	E18	E21	E32	E27	E39		E3	E12	E38	E1
8	+19	+29		E25	E18	E19	E29	E27	E39		E3	E12	E38	E1
9		+22	x30		E18	E19	E22	E27	E39		E3	E12	E38	E1
10	+11	+23		E11	E18	E19	E30	E27	E39	E26	E3		E38	E1
11		+31	x9		E18	E19	E31	E27	E39	E26	E3		E38	E1
12	+41	+28		E41	E18	E19	E31	E9	E28	E26	E3		E38	E1
13	+35	+8	x40	E35	E18	E19	E31			E26	E3	E8	E38	E1
14	+10	+7	x36	E35	E18	E10	E31	E43	E7	E26		E8	E38	E1
15	+15	+42	x6	E35	E18	E15	E36	E43	E39	E26		E8	E38	E1
16		+37	x16	E35	E18	E15	E6	E43	E39	E26		E8	E38	E1
17		+4		E35	E18	E15	E16	E43	E39	E26	E4	E8	E38	
18	+17	+34			E18	E15	E33	E43	E39	E26	E4	E8	E38	
19	+14	+5			E18	E13	E33	E43	E39	E26	E2		E38	

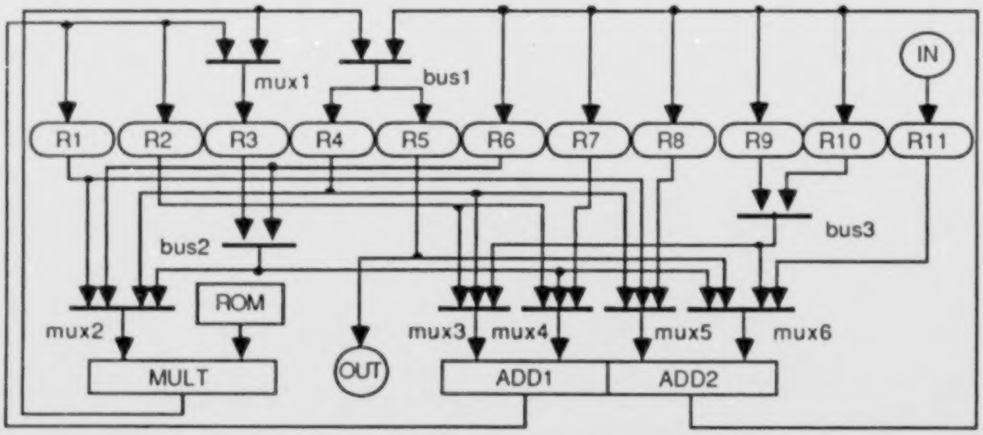
Table 8.6 Schedule and Allocation for the EWF Example

non-optimal schedule, with a cost of 2 adders, 1 multiplier, 10 registers and 8 (instead of the optimum 7) interconnects.

The run times reported in Table 8.5 are misleading in that they may suggest SE-based allocation is slow. In fact, the contrary is true. SE-based allocation obtains good results quickly, but then requires progressively longer run times to improve on these results. Fig. 8.8 plots the cost of the best allocation found as a function of CPU time for the EWF example using 19 control steps and pipelined multipliers. This


bus1
ROM
MULT
R1
R2
R3
R4
R5
R6
R7
R8
mux1
bus2
bus3
R9
R10
OUT
ADD1
ADD2
IN
R11
R12
mux2
mux3

(a) HAL's Circuit


mux1
bus1
IN
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
bus2
bus3
ROM
mux2
mux3
mux4
mux5
mux6
MULT
OUT
ADD1
ADD2

(b) SSE's Circuit

Figure 8.7 Circuits for the EWF Example

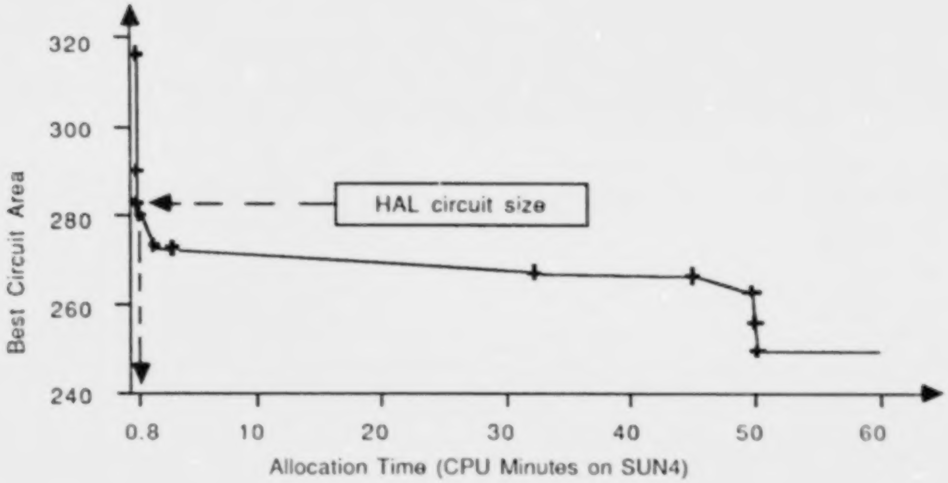


Figure 8.8 Run Time Profile for SE-based Allocation

shows that, while it may take 50 CPU minutes to obtain the best allocation, it only takes around 45 CPU seconds to obtain a design which just surpasses HAL's design.

Finally, we unroll the wave filter CDFG 3 times to create a large example containing 110 nodes and 114 edges. The scheduling performance for this example is summarized in Table 8.7. In [9], this example is scheduled with respect to time and operator cost alone (i.e., ignoring register and interconnect costs), and a result of 50 control steps, 1 pipelined multiplier, 3 adders, and a CPU time of 40 seconds (solving linear programming equations using GAMS/MINOS and ZOOM on a IBM PS/2 model 80) is reported. Considering that our results in Table 8.7 include registers and interconnects, which would have greatly increased the number of linear inequality equations in the model of [9], our scheduling performance compare well with that reported in [9].

Control Steps	Schedule Costs	Iterations	Time (s)
50	3+, 1xP, 10R, 10T	21	22.7
52	3+, 1xP, 10R, 10T	13	16.4
54	3+, 1xP, 10R, 8T	20	34.2
56	2+, 1xP, 11R, 8T	97	134.5
58	2+, 1xP, 11R, 8T	70	116.2
59	2+, 1xP, 10R, 7T	2078	2909.2
50	3+, 2x, 10R, 11T	93	104.8
52	3+, 2x, 10R, 9T	114	137.2
54	3+, 2x, 10R, 11T	185	242.2
56	2+, 2x, 11R, 9T	89	128.1

Table 8.7 Scheduling Performance for the EWF Unrolled 3 Times

8.4. Summary

In this chapter, we have presented experimental results of our SE-based scheduling and allocation algorithms on a number of design examples from the current literature. These results demonstrate that, compared to other synthesis algorithms, SE-based scheduling and allocation algorithms generate comparable designs quickly. More importantly, these results show that, when given longer (but still reasonable) run times, SE-based synthesis can produce better designs than those reported by other synthesis systems. In particular, note that our circuits for the EWF benchmark are the best designs reported to date on this popular benchmark.

Chapter 9. Comparing SE to Simulated Annealing

Simulated Annealing, or SA, is another general combinational optimization algorithm which implements probabilistic search. SA has been successfully applied to a number of optimization problems in the design and synthesis of VLSI circuits [6,40,53,55]. While SA has produced excellent solutions in applications such as standard cell placement [55], it has required considerable CPU run times. On the other hand, recent work on SE [22,29,31,32] has presented benchmark results which suggest that SE is better than SA in terms of solution quality and/or CPU run time. However, we are not aware of any work to date which has investigated why this may (or may not) be so. In this chapter, we will first propose two hypotheses as to why SE may be better than many implementations of SA, especially for highly constrained problems such as scheduling and allocation. We will then describe a number of experiments to test these hypotheses in the context of the scheduling problem, and present results of these experiments to support our intuitive reasoning. Finally, we will describe an algorithm which combines features in both SE and SA. We will present experimental results using this algorithm to provide additional insights to the SE algorithm.

9.1. Simulated Annealing

Recall that in general, a combinatorial optimization problem is specified by a discrete solution space, S, and a cost function, $C(X)$, defined on all solutions $X \subseteq S$, and the optimization objective is to find $X \subseteq S$ which minimizes $C(X)$. In this context, SA is similar to the SE algorithm in that they both implement a probabilistic search over S by randomly moving from one state $i \subseteq S$ to the next state $j \subseteq S$ according to a transition probability $P(i,j)$ [51]. However, SA differs from SE in the way it determines the next state to move to from each current state.

Given a current state, i, the SA algorithm (Fig. 9.1) generates the next state, say j, by randomly applying a transformation, or *move*, to i to obtain a new solution, $k \in S$, which is then probabilistically accepted as j. The acceptance criteria for k are that either $C(k) < C(i)$ (i.e., k is a better solution than i), or $random(0,1) < exp(-(C(k) - C(i))/T)$ where T is "temperature", a control parameter. Initially, the current state is set to a randomly generated solution, and T is assigned a large value so SA has a high probability of accepting more costly solutions as the next states. As SA progresses, the value of T decreases according to a *cooling schedule*, and SA becomes less and less likely to accept more costly solutions as the next states.

```
Algorithm SA;
/* X is the current solution */
/* k is a new solution */
begin
  X := randomly generated initial solution;
  loop until TERMINATE()
    begin
      loop until Equilibrium()
        begin
          k := SA-GENERATE(X);
          if ((C(k) < C(X)) OR (random(0,1) < exp(- (C(k) - C(X))/T)))
          then X := k;
        end loop;
      T := update(T);
    end loop;
  return(X);
end SA;
```

Figure 9.1 Pseudo-Code for the SA Algorithm

Previous work [1, 12, 17, 30, 39, 40, 51] has tried to improve the solution quality and run time efficiency of SA by one of two approaches: move-set design and cooling schedule improvements. In the first approach, carefully selected moves are used to reduce the likelihood of generating solutions which are going to be rejected. Some implementations have combined several simple moves into complex moves to allow faster exploration of the solution space and a higher probability to escape from local minima. Unfortunately, this approach has not been widely used because the move-sets tend to be problem specific and not generally applicable to other optimization problems [17]. As a result, most work has focused on the second approach, in which the annealing process is carefully controlled by selecting the initial temperature, dynamically adjusting the way this temperature is decreased, and defining the termination conditions.

As we will show in this chapter, one reason that SE may be better than SA is that SE uses arbitrarily complex moves whereas most implementations of SA use much simpler moves. We will also describe a general purpose "complex move generator" for SA which is based on the rip-up and reconstruct principles in SE.

9.2. An Intuitive Comparison of SE and SA

Comparing the SE and SA algorithms, we make two intuitive arguments as to why SE may be better than many implementations of SA. First, while both SE and SA implement a probabilistic search over the solution space, the transition probability, $P(i,j)$, in SA is a simple function of $C(i)$, $C(j)$ and T, whereas $P(i,j)$ in SE depends on the application specific heuristics and cost functions in the *GENERATE* and *SELECT* steps. Intuitively, SE should be better than SA because it implements a more guided search than SA.

Second, constructing a new solution from parts of the current solution allows SE to make arbitrarily *distant* state transitions (as measured by the number of solution elements that are different between consecutive states). This contrasts with most SA implementations which use short-distance moves (e.g., changing the value of a single solution element, and interchanging the values of two solution elements). Intuitively, the distant state transitions allow SE to cover the solution space more effectively than SA. This is especially pronounced in highly constrained problems such as the scheduling problem, in which most solutions produced by randomly applied simple moves are either *illegal* or higher cost solutions. In this case, SA tends to spend a lot of time generating new solutions and then rejecting them, hence SA has a higher probability of being trapped in local minima for long periods of time than SE.

To test the above hypotheses, we designed two sets of experiments in the context of the scheduling problem. Basically, we separated heuristics guided search and state transition distance, and tried to measure the effects of each on the performance of probabilistic search. The next two sections present these experimental results.

9.3. Experiments on Effects of Guided Search

In the first set of experiments, we measure the effects of search heuristics using our SE-based scheduling algorithm. Starting with the SE algorithm, we successively replace each of the cost functions (i.e., one at a time) *PRIORITY*, *INCR* and *GLOBAL* by a random number generator, and compare the statistical performance of each of these "randomized" scheduling algorithms against that of the original SE-based scheduling algorithm.

The statistical performance of a scheduling algorithm is measured as follows. Given a CDFG, we run the scheduling algorithm for a maximum of, say, 100 iterations, terminating as soon as a schedule is found whose cost is less than or equal to a target cost. This is repeated for a total of 30 runs, and the last iteration number in each run is recorded. At the end of 30 runs, we compile the recorded iteration numbers into a frequency histogram, which represents the statistical distribution of the number of iterations required to generate the target schedule. This frequency histogram then gives a measure of the statistical performance of the scheduling algorithm.

For the fifth order Elliptic Wave Filter (EWF) example with a global timing constraint of 19 control steps and using pipelined multipliers, the best schedules have a cost of 2 adders, 1 multiplier, 10 registers and 7 interconnects.

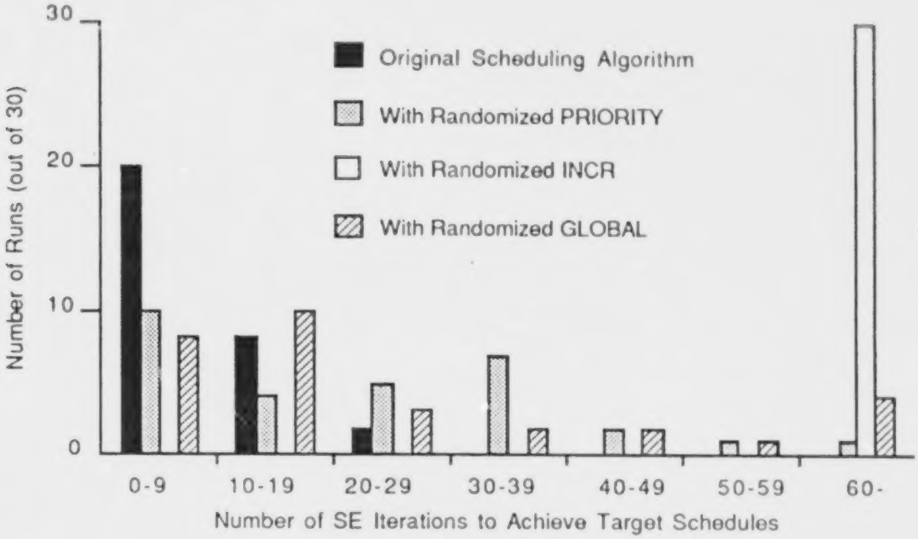


Figure 9.2 Performance with Randomized PRIORITY, INCR and GLOBAL

Fig. 9.2 shows the statistical performance of each randomized scheduling algorithm for this example using a target cost of 2 adders, 1 multiplier, 10 registers and 8 interconnects (i.e., just slightly over the optimum). As expected, scheduling performance deteriorates when each of the cost functions is randomized. Moreover, these results clearly show that the performance of the SE algorithm depends primarily on *INCR*, and then, to a much lesser extent, on *GLOBAL* and *PRIORITY*, in that order.

To gain further insight into the effects of *INCR* on the performance of SE-based scheduling, we remove in turn the component costs in *INCR* and compare the statistical performance of each of these scheduling algorithms. Recall that (Chapter 5) *INCR* in our SE-based scheduling algorithm consists of four component costs: I_{OP}, which calculates the incremental operator cost; I_R, which calculates the incremental register cost; I_T, which calculates the incremental interconnect cost; and I_P, which calculates the incremental opportunity cost due to the decreased freedoms in the predecessor and successor nodes.

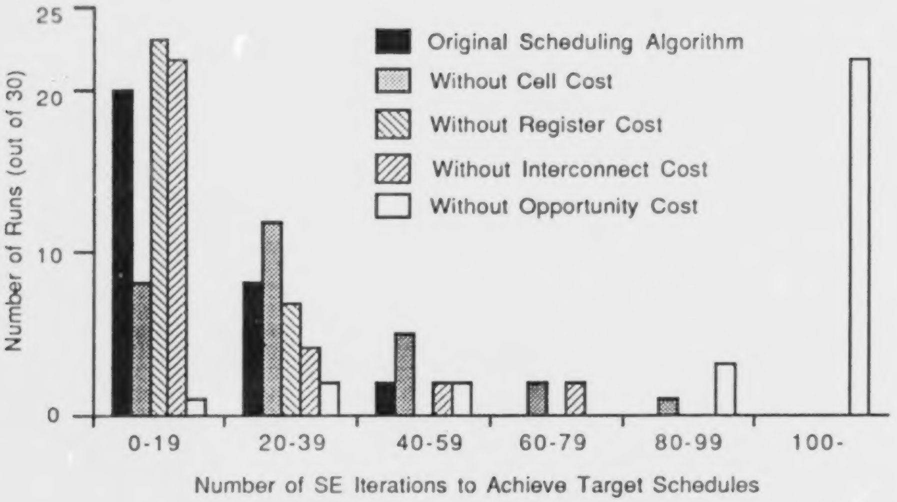


Figure 9.3 Performance Without Component Costs in INCR (Case 1)

Fig. 9.3 shows the results of these experiments, which clearly show that the scheduling performance depends primarily on I_P, and then, to a much lesser extent, on I_{OP}. It is interesting to note that the removal of I_T has only a minor effect on the scheduling performance, and the removal of I_R actually *improves* the statistical performance of the SE-based scheduling algorithm.

To verify this, we repeat the above experiments with the optimum target cost of 2 adders, 1 multiplier, 10 registers and 7 interconnects, and ran for a maximum of 300 SE iterations. Fig. 9.4 presents the results of these experiments, which show the same relationship between the statistical scheduling performance and the cost functions I_P and I_{OP}. However, they also show that removing I_T severely degrades the scheduling performance, and removing I_R also slightly degrades scheduling performance. This illustrates that, for this particular benchmark, it is easy to achieve a near optimum schedule which requires 2 adders, 1 multiplier, 10 registers and 8 interconnects, but

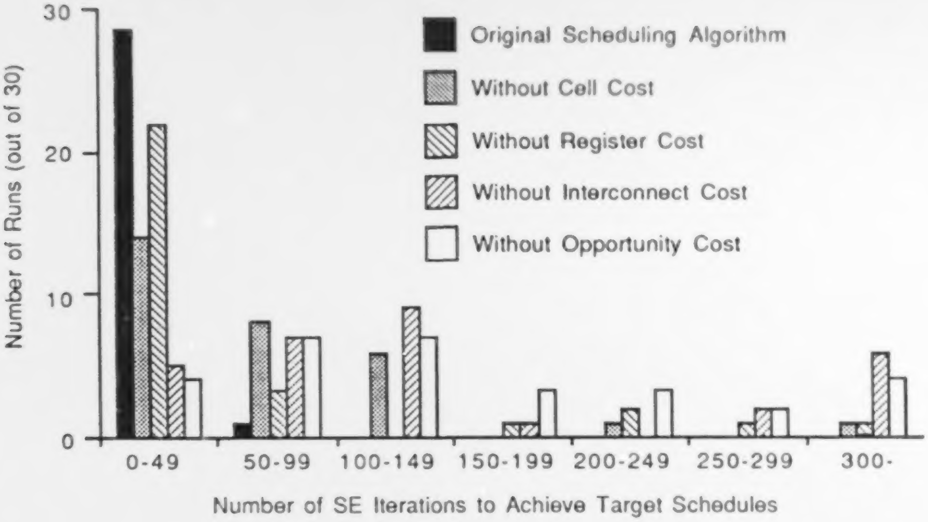


Figure 9.4 Performance Without Component Costs in INCR (Case 2)

much more difficult to achieve the optimum schedule requiring only 7 interconnects.

We draw three conclusions from the above results. First, the performance of SE-based scheduling derives primarily from its ability to make locally optimum decisions during greedy scheduling (i.e., *INCR*). Second, the performance of the greedy scheduling algorithm itself depends primarily on the incremental opportunity cost, I_P (which essentially implements a one step lookahead for the algorithm). Third, for near optimum schedules, removing certain cost functions (i.e., I_T and I_R) not only may not degrade, but may actually improve, the scheduling performance. However, for optimum schedules, removing any of the cost components in *INCR* definitely degrades the performance of SE-based scheduling. This shows that a more random search may be better at obtaining *good* schedules since there may be many such schedules, but a directed search is still better at obtaining the optimum schedules.

9.4. Experiments on Effects of State Transition Distance

In the second set of experiments, we measure the effects of state transition distance using a SA-based scheduling algorithm which generates a new solution from a current solution by randomly de-scheduling up to M CDFG nodes, and then randomly scheduling these nodes (see Fig. 9.5). Since at most M nodes may be different between consecutive schedules, the parameter M controls the *maximum* state transition distance in this SA algorithm. This SA implementation is very similar to other SA-based algorithms if M is small (e.g., 2 or 3), but becomes an *unguided* version of SE-based scheduling if M is set to the number of nodes in the CDFG (i.e., N). Consequently, we may measure the effects of (maximum) state transition distance by comparing the statistical performance of this SA-based scheduling algorithm for different values of M.

```
Function SA-GENERATE(X);
  begin
    /* M is the maximum state transition distance allowed*/
    /* d is a randomly generated state transition distance */
    /* k is the new schedule generated from the current schedule, X */
    d := random(1,M);
    k := X;
    casualties := randomly pick d nodes in k;
    de-schedule all nodes in casualties from k;
    for each node n in casualties do
      begin
        candidates := determine feasible schedules for n;
        update k;
        t := randomly pick a control step from candidates;
        assign t to the schedule of n in k;
      end;
    return(k);
  end;
```

Figure 9.5 Pseudo-Code for SA-GENERATE

For the EWF example with 19 control steps and using pipelined multipliers, we measure the statistical performance of this SA-based scheduling algorithm by running a maximum of 1200 iterations for 30 runs (starting with the same, randomly generated initial schedules), and with a target cost of 2 adders, 1 multiplier, 10 registers and 8 interconnects (again, near optimum).

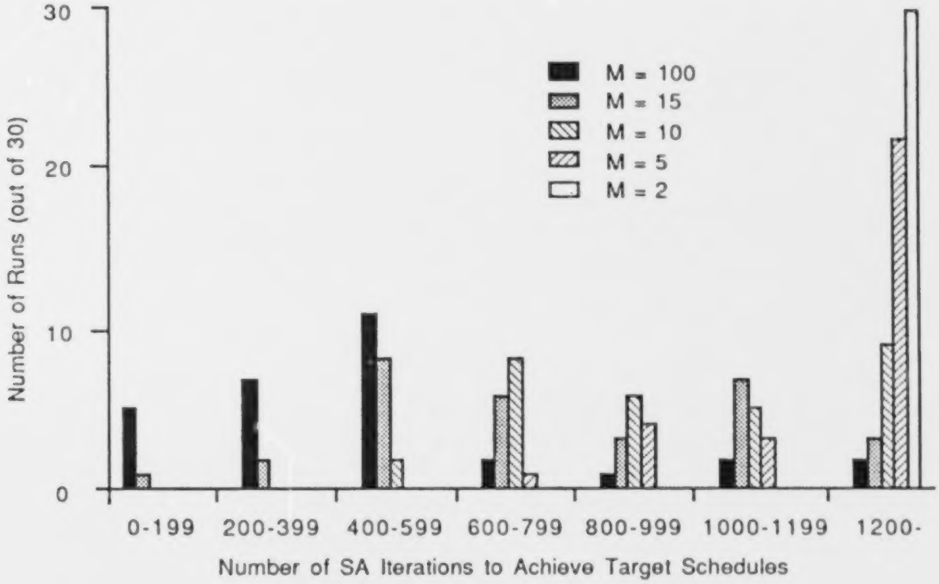


Figure 9.6 Statistical Performance for SA-Based Scheduling

Fig. 9.6 compares the statistical scheduling performance for this case for a few representative values of M. We also measure the scheduling performance for the case of 19 control steps, using non-pipelined multipliers, and with a target cost of 2 adders, 2 multipliers, 10 registers and 9 interconnects (the optimum schedules for this case use 1 less interconnect). Fig. 9.7 plots the number of runs in which SA-based scheduling *failed* to find the target schedules as a function of M for both of these cases (the EWF example has 38 nodes). These results clearly show that the performance of probabilistic search improves with increasing maximum state transition distance.

Next, we modify *SELECT* to de-schedule a maximum of M nodes so as to measure the effects of (maximum) state transition distance on the SE algorithm. For exam-

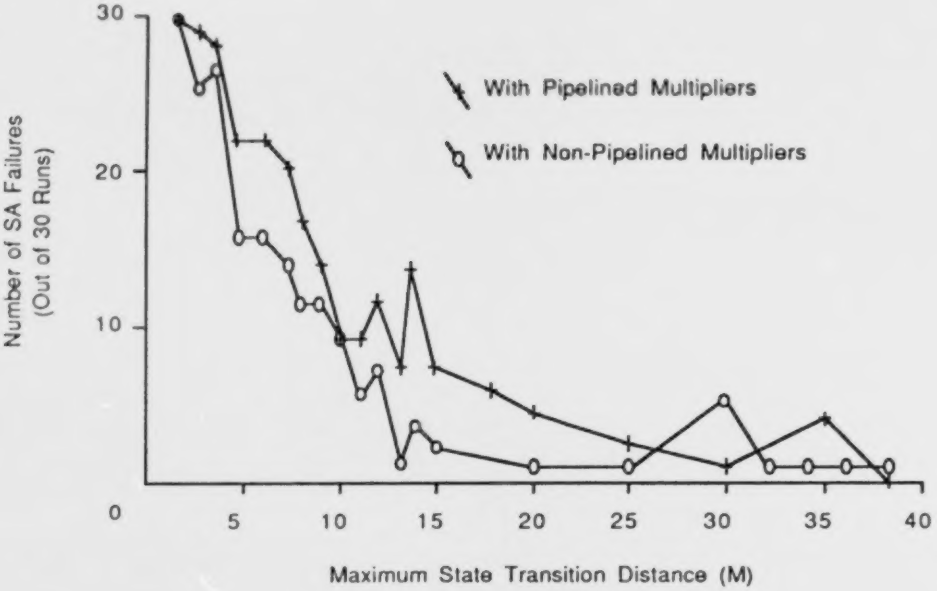


Figure 9.7 SA Failures vs. Maximum State Transition Distance

ple, if $M = 5$, and the number of nodes originally selected for de-scheduling is greater than 5, then we only de-schedule the 5 most costly of the selected nodes. For the EWF example with 19 control steps and using pipelined multipliers, we measure the statistical performance of this SE-based scheduling algorithm by running a maximum of 70 iterations for 30 runs, with a target cost of 2 adders, 1 multiplier, 10 registers and 8 interconnects. We also measure the scheduling performance for the case of $G = 19$, using non-pipelined multipliers, with a target cost of 2 adders, 2 multipliers, 10 registers and 9 interconnects, and running a maximum of 100 iterations. Fig. 9.8 plots the number of runs in which SE-based scheduling failed to find the target schedules as a function of M for both cases. These results clearly show the importance of distant state transitions in the statistical performance of SE.

We draw two conclusions from the above results. First, for highly constrained

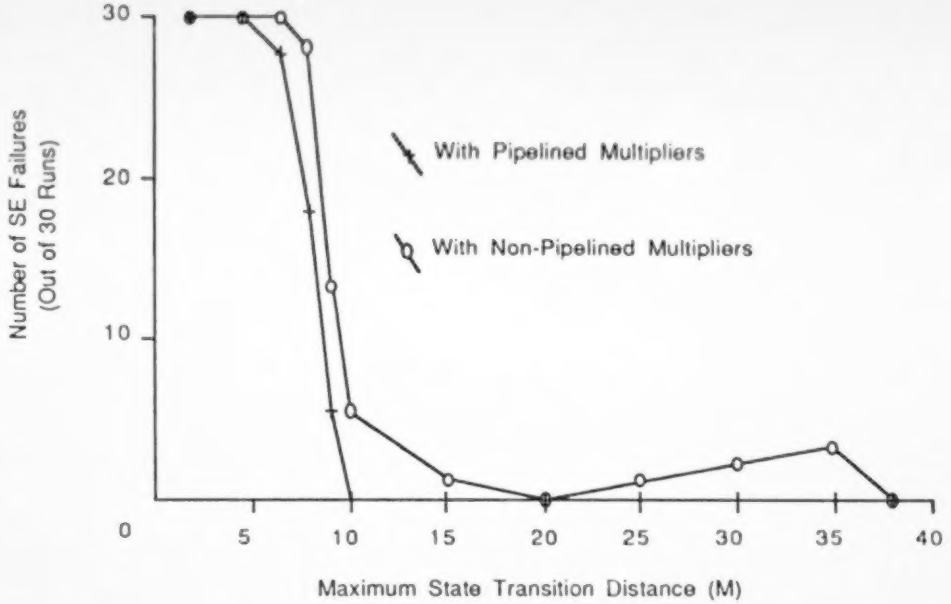


Figure 9.8 SE Failures vs. Maximum State Transition Distance

problems like scheduling, the SE algorithm should be better than many implementations of SA because SE permits arbitrarily distant state transitions (i.e., $M = N$). Second, we may implement SA to allow arbitrarily distant state transitions and therefore improve the performance of SA. Such an implementation may actually outperform SE if the larger numbers of SA iterations are more than offset by the increase in iteration speed as a result of not having to calculate the cost functions *PRIORITY*, *INCR* and *GLOBAL*.

9.5. Combining the SE and SA Algorithms

Finally, we combine the SE and SA algorithms by modifying SE to probabilistically accept a new solution as the next state using SA's acceptance criteria (Fig. 9.9). For the EWF example with $G = 19$ and using pipelined multipliers, we measure the statistical performance of this algorithm by running a maximum of 30 iterations for 30 runs (starting with the same schedules produced by *GENERATE* ($\emptyset$)), and with a target schedule of 2 adders, 1 multiplier, 10 registers and 8 interconnects. Fig. 9.10 shows the statistical performance of this algorithm for several representative values for the initial temperature, T_0. Basically, the higher T_0 is, the more likely the algorithm is to accept more costly solutions as the next states in the initial iterations. For example, setting $T_0 = 1000$ effectively reduces the algorithm to SE because every solution generated is accepted as a next state. On the other hand, setting $T_0 = 0$ reduces the algorithm to a greedy search because only solutions which are cheaper than the current states are accepted as the next states. For the intermediate cases (i.e., $0 < T_0 < 1000$), the algorithm behaves as SA with different probabilities for accepting hill climbing moves.

The results in Fig. 9.10 clearly show that the performance of this algorithm does not depend on the cooling schedule since, regardless of T_0, most of the runs have obtained the target schedules within 10 iterations, before T has even started to decrease by any appreciable amount. Moreover, the performance of this algorithm for different values of T_0 is so statistically similar that they suggest the SE part of this algorithm is largely responsible for its performance. (Although Fig. 9.10 seems to show an improvement in scheduling performance as T_0 decreases, a closer examination of the raw data revealed that this is primarily a result of the histogram intervals used in the plot, and we cannot infer that lowering T_0 produces better statistical performance. On the other hand, we can infer that lowering T_0, even to 0, does not necessarily degrade the performance for the SE/SA algorithm.)

```
Algorithm SE-SA;
/* X is the current solution */
begin
  X := GENERATE(∅);
  loop until TERMINATE()
    begin
      T := $T_0$;
      loop until Equilibrium()
        begin
          /* p is a partial solution */
          /* k is a new solution */
          p := SELECT(X);
          k := GENERATE(p);
          if ((C(k) < C(X)) OR (random(0,1) < exp(- (C(k) - C(X))/T)))
          then X := k;
        end loop;
      T := update(T);
    end loop;
  return(X);
end SE-SA;
```

Figure 9.9 Pseudo-Code for the Combined SE/SA Algorithm

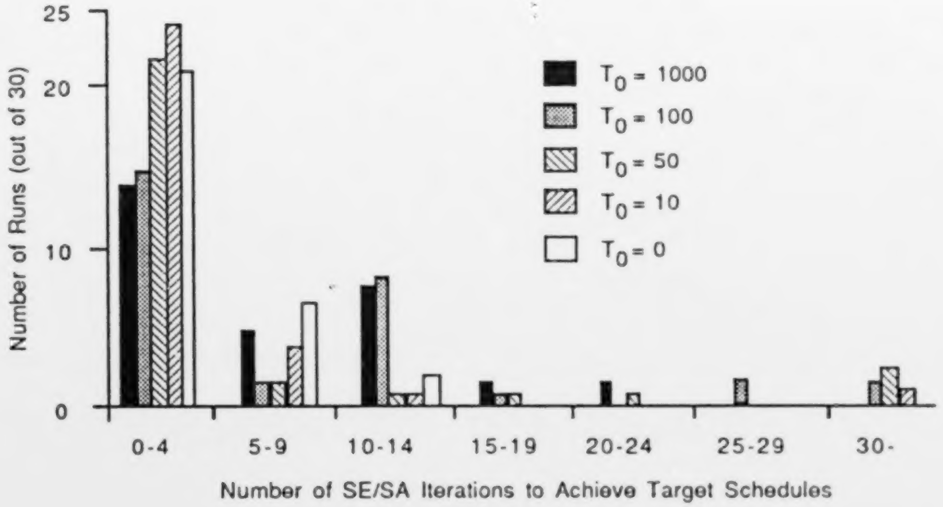


Figure 9.10 Performance of SE/SA Based Scheduling for Select T_0 Values

We draw two conclusions from these (and other similar) experimental results. First, since the *GENERATE* and *SELECT* steps implement arbitrarily distant jumps in the solution space, the SE algorithm does not benefit from a cooling schedule as in the case of SA. Second, the performance of SE is not significantly affected by various acceptance rules for new states. In particular, the statistical performance for $T_0 = 0$ shows that *GENERATE* and *SELECT* are capable of achieving an effective exploration of the solution space even with a greedy acceptance rule.

9.6. Summary

In summary, we have presented in this chapter experimental results which demonstrate that, for highly constrained optimization problems such as the task of scheduling, SE may be better than many implementations of SA for two reasons. First, the use of application specific heuristics and cost functions in SE allows it to implement a more directed search than SA. In particular, its ability to find locally optimum solutions in a subspace of the solution space appears to be the key to SE's performance. Second, the rip-up and reconstruct steps in SE implements much more distant state transitions than allowed by most implementations of SA. For highly constrained problems like scheduling, being able to make arbitrarily distant state transitions evidently allows SE to explore the solution space much more effectively than SA. On the other hand, the SA algorithm can be implemented so as to allow arbitrary state transitions. In particular, the *SA−GENERATE* function in Fig. 9.5 outlines a general purpose "complex move generator" for SA which is based on the rip-up and reconstruct principles in SE. Whether such an implementation would be better than SE then depends on whether the faster SA iteration speed can make up for the larger numbers of SA iterations required to achieve the same solution quality as SE. In our implementation, the SE-based scheduling algorithm consistently outperforms SA-based scheduling (with maximum state transition distance) in CPU times by a factor of 10. Other SA-based scheduling algorithms [6, 40] have achieved more reasonable CPU performance on the EWF example using better cooling schedules, more efficient representations, and by coding in C (instead of LISP).

Finally, we have combined SE and SA into a single algorithm which probabilistically accepts as next states the new solutions generated by *SELECT* and *GENERATE*. Using this algorithm, we have presented experimental results which show that the SE algorithm does not benefit from a cooling schedule, and that the statistical performance of SE is not significantly affected by various acceptance rules for new states.

Chapter 10. Bottom Up Synthesis based on Fuzzy Schedules

Throughout chapters 4 to 9, we have assumed that all CDFG nodes are scheduled prior to allocation. In this chapter, we remove this assumption by extending our allocation algorithm to accommodate partial or, more appropriately, *incomplete* schedules. To do so, we will first describe a new formulation of the allocation problem designed for *fine-grained bottom-up synthesis*, in which we assume that *all CDFG nodes and edges are allocated before scheduling*. This approach to bottom-up synthesis is novel because it treats unscheduled nodes as having *uncertain* schedules, and applies fuzzy set theory to quantify these uncertainties during allocation. We will then discuss the implementation of this formulation as simple extensions to our SE-based allocation algorithm, and present experimental results which demonstrate the effectiveness of this approach. We will also describe how our scheduling algorithm is extended to accept a partially allocated CDFG.

10.1. Introduction

Scheduling and allocation are highly interdependent tasks. Basically, two or more operations, data values, or data transfers cannot be implemented on the same hardware resource at the same time. (Mutual exclusion is ignored for now to simplify this discussion.) Therefore, during scheduling, we need to know if two nodes or edges will be allocated to the same cells before we schedule them to have overlapping lifetimes; during allocation, we need to know if two nodes or edges will be scheduled to have overlapping lifetimes before we allocate them to the same cells. This leads to a vicious cycle: scheduling depends on the results of allocation, which in turn depends on the results of scheduling.

Previous synthesis systems have approached the tasks of scheduling and allocation in one of three ways: global, top-down and bottom-up. In the global approach, scheduling and allocation are treated as a single task. In the top-down approach, scheduling is performed first, followed by allocation. In the bottom-up approach, allocation is performed first, followed by scheduling. Most synthesis systems to date have taken either the global approach or the top-down approach. The global approach is taken by many transformational synthesis systems (see Section 3.3), such as DAA [25] and CAMAD [49], in which an initial design is successively modified by design transformations which may change both the schedules and allocations in the designs. On the other hand, the top-down approach is taken by most constructive synthesis systems [37], in which allocation is performed either after scheduling (e.g., HAL [48]), or concurrently with scheduling to provide cost estimates for the scheduling algorithm (e.g., ELF [11]).

In contrast, the bottom-up approach has only recently received some attention. The BUD/DAA [38] system incorporates bottom-up design techniques by performing *design partitioning* prior to scheduling and final allocation. BUD uses a hierarchical clustering method to group CDFG nodes into clusters based on a distance metric that weighs potential hardware sharing, interconnect, and parallelism between each pair of nodes. These clusters then provide cost estimates for low level characteristics, such as wire lengths and layout areas, during scheduling. Different granularities of design partitions are tried so as to find the best designs. This work has been extended by Scheichenzuber et. al. [54], who defined a more extensive set of distance metrics between CDFG nodes, including a *schedule distance* that takes into account the *probability* of scheduling conflicts between every pair of nodes. Note that design partitioning is different from allocation since partitioning only produces subcircuits which may each contain multiple functional units and registers. These systems can be viewed as implementing *coarse-grained* bottom-up synthesis, as opposed to *fine-grained* bottom-up synthesis in which full allocation of all CDFG nodes and edges is performed prior to scheduling.

In this chapter, we present a new formulation of the allocation task designed for fine-grained bottom-up synthesis. Basically, we view unscheduled CDFG nodes as having *uncertain* schedules. We know that each node will be assigned to one of its candidate control steps, but we do not know which one until the node is eventually scheduled. In the mean time, we may estimate the relative merit of the scheduling candidates and apply fuzzy set theory [24] to represent our *beliefs* that some control steps are better scheduling candidates than others. This leads to the concept of *fuzzy schedules* for CDFG nodes. From these fuzzy schedules, we derive the concept of *fuzzy lifetimes* for all CDFG nodes and edges, and then calculate a *fuzzy schedulability* which measures the belief that a feasible schedule still exists given the allocation. This allows allocation to trade off hardware cost and the *perceived risk of scheduling conflicts*. Consequently, we generalize the task of allocation to that of finding the smallest circuit for which a *feasible schedule is still highly possible*. We describe our implementation of these concepts as simple extensions to our SE-based allocation algorithm, and present experimental results to demonstrate the effectiveness of this approach for bottom-up synthesis.

The organization of this chapter is as follows. In Section 10.2, we define the concepts of fuzzy schedule, fuzzy lifetime, and fuzzy schedulability, and we describe how incremental changes in fuzzy schedulabilities can be equated with circuit area to trade off hardware cost and the risk of scheduling conflicts. Section 10.3 discusses a number of implementation issues which arise when extending our allocation algorithm to handle fuzzy schedules. Section 10.4 presents experimental results which demonstrate the strength of fuzzy allocation (i.e., allocation with fuzzy schedules), and evaluates different fuzzy allocation schemes based on these results. Section 10.5 compares fuzzy allocation and design partitioning, discusses alternate formulations of risk of scheduling conflicts, and describe how we extend our scheduling algorithm to make effective use of the allocation information in the CDFG. Finally, Section 10.6

summarizes this chapter.

10.2. Fuzzy Schedulability

Resource contention conflicts occur when CDFG nodes or edges with overlapping lifetimes are allocated to the same cells. In top-down synthesis, CDFG nodes and edges have well defined lifetimes when they are allocated, so resource conflicts are avoided by ensuring that allocation candidates exclude cells which are already allocated to other nodes or edges with overlapping lifetimes. In bottom-up synthesis, however, the lifetimes for CDFG nodes and edges are ill-defined during allocation, so a new way of evaluating the risk of resource conflicts is needed. Toward this end, we develop the concept of *fuzzy schedulability*, which quantifies the belief that feasible schedules are still possible given the allocation. This allows the allocation algorithm to consider the risk of resource conflicts in the face of uncertain schedules.

In this section, we first give an overview of fuzzy set theory. We then define the concepts *fuzzy schedule*, *fuzzy lifetime*, and *fuzzy schedulability*. Finally, we equate changes in fuzzy schedulabilities with circuit area, and describe how these concepts are incorporated into our allocation algorithm.

10.2.1. Fuzzy Set Theory

In classical set theory, an element, e, in the universe of discourse, U, either lies in a set $A \subseteq U$ or lies outside the set. This property of classical sets may be represented by the membership function of the set A, μ_A:

$$\mu_A(e) = \begin{cases} 1 & \text{if } e \in A \\ 0 & \textit{otherwise} \end{cases}$$

However, in fuzzy set theory, the 0/1 value restriction on the membership function of sets is relaxed. Specifically, the membership function, $\mu_F(e)$, for a fuzzy set F takes values on any real number from 0 to 1 inclusive (i.e., $0.0 \le \mu_F(e) \le 1.0$). $\mu_F(e) = 1$ means e is definitely in the set F; $\mu_F(e) = 0$ means e is definitely *not* in the set F; and $0 < \mu_F(e) < 1$ means e is *possibly* in the set F. The higher the value of $\mu_F(e)$, the more *confident* we are about the membership of e in F. In essence, $\mu_F(e)$ specifies a subjective *degree of confidence* about the membership of e in F. This is useful for modeling uncertainties which are not random in nature [24].

Three basic operators on fuzzy sets are set union ($A \cup B$), set intersection ($A \cap B$), and set complement ($\bar{A}$). These operations may be defined in terms of operations on the membership functions of their operands as follows [24]:

$$\mu_{A \cup B}(e) = Max(\mu_A(e)\ ,\ \mu_B(e)),$$
$$\mu_{A \cap B}(e) = Min(\mu_A(e)\ ,\ \mu_B(e)),$$
$$\mu_{\bar{A}}(e) = 1 - \mu_A(e).$$

For the purpose of this thesis, we define a *fuzzy variable* as a countable fuzzy set in the universe of positive integers. Moreover, to simplify notations, we denote the membership function, $\mu_V(i)$, of a fuzzy variable, V, by a function of the same name (i.e., $V(i)$).

10.2.2. Fuzzy Schedule

We define the fuzzy schedule, S_n, for a CDFG node, n, as a fuzzy variable which represents the set containing the control step that n will be scheduled to. The membership function, $S_n(i)$, then represents our subjective belief that n will be scheduled to start in control step i in the final design. For example, if n is already scheduled to a control step x, then $S_n(x) = 1$ and $S_n(i) = 0$ for all $i \ne x$. On the other hand, if n is unscheduled, then $0 \le S_n(i) < 1$ for $asap(n) \le i \le alap(n)$ and $S_n(i) = 0$ for all other values of i, where $asap(n)$ and $alap(n)$ denote the *as soon as possible* and the *as late as possible* schedules for n, respectively.

Since every CDFG node n must be scheduled in one and only one control step between $asap(n)$ and $alap(n)$, we can reduce $S_n(i)$ to a *probability measure* (i.e., a function which satisfies the Axioms of Probability [24]) by imposing the condition:

$$\sum_i S_n(i) = 1.0 \, . \tag{10.1}$$

Then $S_n(i)$ becomes a *probability of belief.*

Unlike statistical probability which is a probability measure induced on some random event, probability of belief is a probability measure induced on a predicate [14]. For example, the weather forecast "30% chance of rain" expresses a probability of belief (i.e., we believe with 30% certainty that it will rain) based on analysis of similar historical weather patterns. In the case of fuzzy schedules, $S_n(i)$ can be interpreted as the probability that the statement "CDFG node n will be scheduled to control step i" will be true in the final design.

Given two probability measures $A(i)$ and $B(i)$, we can write:

$$(A \cup B)(i) = A(i) + B(i) - (A \cap B)(i)$$

If $A(i)$ and $B(i)$ are disjoint probabilities, then $(A \cap B)(i) = 0$ and hence $(A \cup B)(i) = A(i) + B(i)$. On the other hand, if $A(i)$ and $B(i)$ are independent probabilities, then $(A \cap B)(i) = A(i) \times B(i)$ and $(A \cup B)(i) = A(i) + B(i) - A(i) \times B(i)$. However, if $A(i)$ and $B(i)$ are neither disjoint nor independent, then we do not have sufficient information to compute either $(A \cup B)(i)$ or $(A \cap B)(i)$. In this case, instead of blindly assuming that $A(i)$ and $B(i)$ are independent, we treat $A(i)$ and $B(i)$ as the membership functions for the fuzzy variables A and B, and apply fuzzy set operations to derive the membership functions for the fuzzy variables $A \cup B$ and $A \cap B$.

Ideally, for an unscheduled CDFG node, n, the values $S_n(i)$ should reflect the *strengths of evidence* that the scheduling algorithm will favor control step i over other candidate control steps when scheduling the node n. However, it is difficult to analytically predict the behavior of a scheduling algorithm, so we use a more practical measure based on dependency analysis of the CDFG. Basically, a node n with a delay of $d(n)$ control steps will be scheduled to control step i only if all of its predecessor nodes, $pred(n)$, will end before control step i, and if all of its successor nodes, $succ(n)$, will start after control step $i+d(n)-1$. Therefore we iteratively adjust the fuzzy schedule of each CDFG node according to the fuzzy schedules of its predecessor and successor nodes.

We start with *ignorance* about the scheduling algorithm, and assume equal probabilities of belief (or simply, beliefs) for all scheduling candidates for each node n:

$$S_n(i) = \frac{1}{alap(n) - asap(n) + 1} \qquad \text{for } i = asap(n) \cdots alap(n).$$

Given these initial values for $S_n(i)$, we define $eb_n(i)$ as the belief that n will *end before* control step i. This is calculated as the belief that n will be scheduled in any control steps from $asap(n)$ to $i - d(n)$:

$$eb_n(i) = \sum_{t=asap(n)}^{t=i-d(n)} S_n(t). \tag{10.2a}$$

Similarly, we define $sa_n(i)$ as the belief that n will *start after* control step i. This is calculated as the belief that n will be scheduled in any control steps from $i+1$ to $alap(n)$:

$$sa_n(i) = \sum_{t=i+1}^{alap(n)} S_n(t) \tag{10.2b}$$

Note that in defining the fuzzy variables eb_n and sa_n, the belief of a CDFG node n being scheduled in *either* control step i or j $(i \neq j)$ is calculated as the sum of beliefs, $S_n(i) + S_n(j)$. This is because $S_n(t)$ is a probability distribution function due to condition (10.1), and that n can never be scheduled to *both* control steps i and j in the final design (i.e., $S_n(i)$ and $S_n(j)$ are disjoint probabilities whenever $i \neq j$).

Given the above definitions, we update the fuzzy schedule of each node n to be the conjunction of eb_p for all $p \in pred(n)$ and sa_s for all $s \in succ(n)$:

$$S_n(i) \leftarrow Min(\underset{p \in pred(n)}{Min} eb_p(i) , \underset{s \in succ(n)}{Min} sa_s(i+d(n)-1)). \qquad (10.3)$$

In this case, we cannot apply simple probability theory because we lack information to calculate the joint probabilities involving different probability functions. For example, to calculate the probability that two CDFG nodes, n and m, will both end before control step i, we need to know the joint probabilities between the scheduling of n and m. In [54], this problem is set aside by (implicitly) assuming that nodes n and m are scheduled independently, hence the joint probability is calculated by $eb_n(i) \times eb_m(i)$. However, this is not a satisfactory solution since the basic assumption (that CDFG nodes are scheduled independently of one another) seriously under-estimates the ability of the scheduling algorithms.

Consequently, we interpret $eb_p(i)$ and $sa_s(i)$ not as probabilities, but as subjective beliefs (see [24], in which probability theory is shown as a subset of belief theory) which we assign as the membership functions of the fuzzy variables eb_p and sa_s, and calculate the new S_n as a conjunction of these fuzzy variables. Specifically, we successively adjust the value of S_n by iterating equations (10.2) and (10.3) for a user-specified number of times (e.g., 3). Basically, each iteration adjusts the value of S_n to take into account the fuzzy schedules for ancestor and descendent nodes which are more and more distant from n.

The actual calculation of S_n is complicated by a normalization step which ensures that condition (10.1) is satisfied after each update, and by a weight function which uses the hardware costs of nodes to determine the relative importance among predecessor and successor nodes:

$$S_n(i) \leftarrow N(S_n) \times Min(\underset{p \in pred(n)}{Min} w(n,p,eb_p(i)) , \underset{s \in succ(n)}{Min} w(n,s,sa_s(i+d(n)-1))) \qquad (10.4)$$

where $N(S_n)$ is a normalization factor computed as:

$$N(S_n) = \frac{1}{\sum_i Min(\underset{p \in pred(n)}{Min} w(n,p,eb_p(i)) , \underset{s \in succ(n)}{Min} w(n,s,sa_s(i+d(n)-1)))}$$

and $w(n_i, n_j, f)$ is a weight function which adjusts the belief value f $(0.0 < f < 1.0)$ according to the relative cell costs for nodes n_i and n_j:

$$w(n_i, n_j, f) = \begin{cases} (c(op(n_i))/c(op(n_j)))\times f & \text{if } c(op(n_i)) \le c(op(n_j)) \\ 1.0 - (c(op(n_j))/c(op(n_i)))\times(1.0 - f) & \textit{otherwise} \end{cases}$$

That is, we give more weight to eb_p (or sa_s) in equation (10.4) if the predecessor node p (successor node s) has a larger cell cost than n by *decreasing* the values $eb_p(i)$ $(sa_s(i+d(n)-1))$; and we give less weight to eb_p (or sa_s) in equation (10.4) if the predecessor node p (successor node s) has a smaller cell cost than n by *increasing* the values $eb_p(i)$ $(sa_s(i+d(n)-1))$. This unusual weight function is due to the *Min* function in equation (10.4). (Note that we must decrease the value of A if we want to increase the importance of A in $Min(A, B)$.)

10.2.3. Fuzzy Lifetimes

Given the fuzzy schedules, we may calculate the lifetime of every CDFG node and edge as a fuzzy variable. More formally, we define the fuzzy lifetime, $live_x$, for a CDFG element (i.e., node or edge), x, as a fuzzy variable which represents the set containing all control steps in which x is *live*. Intuitively, $live_x(i)$ measures the belief that control step i will lie in the lifetime of x in the final design. For a CDFG node n with a latency of $l(n)$ control steps (recall that $l(n) = d(n)$ for non-pipelined operations), n is *live* in control step i if it starts execution in any control steps from $i-l(n)+1$ to i. Thus $live_n(i)$ is calculated as the sum of beliefs for n being scheduled to control steps from $i-l(n)+1$ to i:

$$live_n(i) = \sum_{t=i-l(n)+1}^{i} S_n(t).$$

Note that if n is a single cycle operation or a pipelined operation with latency 1, then $live_n$ reduces to the fuzzy schedule S_n.

On the other hand, the lifetime of a CDFG edge, e, starts in the control step in which the node generating e ends, and ends in the control step just before the last access of e. For example, if the last of the nodes accessing e ends (or can start a new pipe-stage) in control step i (i.e., e is last accessed in i), then e must be stored in a register until control step $i-1$, hence its lifetime ends in control step $i-1$. More formally, if we denote by $src(e)$ the node generating e, and denote by $dests(e)$ the set of nodes accessing e, then we define $live_e(i)$ as:

$$live_e(i) = Min(eb_{src(e)}(i+1) \ , \ \underset{n \in dests(e)}{Max} sa_n(i-l(n)+1)).$$

That is, an edge e is live in control step i if and only if the node $src(e)$ ends before control step $i+1$ (i.e., ends in or before i), and at least one node in $dests(e)$ will still access e after control step i. (A node n will access its inputs after a control step i if n starts after the control step $i - l(n) + 1$). Again, fuzzy set operators are used in the above equation because $eb_{src(e)}(i+1)$ and $sa_n(i+l(n)-1)$ ($n \in dests(e)$) are obviously interdependent probabilities.

To illustrate the above calculations, consider the CDFG segment in Fig. 10.1 (taken from the EWF CDFG in Fig. 8.6). We assume that additions are single cycle operations, multiplications are 2-cycle operations (and not pipelined), and that fuzzy schedules for nodes +27, +29 and +30 are assigned values shown in Fig. 10.2. Fig. 10.3 shows the fuzzy lifetimes for these nodes. The fuzzy lifetimes for +27 and +29 are identical to their fuzzy schedules, and the fuzzy lifetime for +30 is calculated by:

$$live_{+30}(i) = S_{+30}(i-1) + S_{+30}(i)$$

since $l(+30) = 2$. Fig. 10.4 shows the *ended before* fuzzy variables, and Fig. 10.5 shows the *started after* fuzzy variables for these nodes. Finally, Fig. 10.6 shows the fuzzy lifetimes for edges $e27$ and $e29$. Notice that the lifetime for $e27$ is determined by the node +28 (and not by +29) because +28 is the last node accessing $e27$.

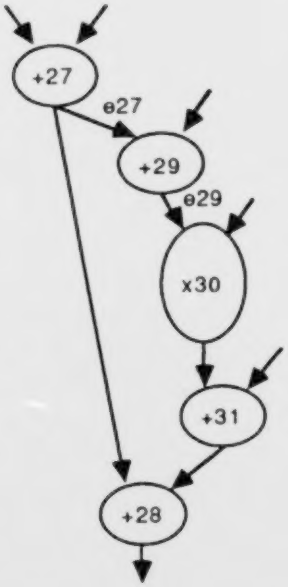


Figure 10.1 CDFG Segment from the EWF Example

CDFG Nodes	8	9	10	11	12
+27	0.01	0.33	0.66	0.00	0.00
+29	0.00	0.66	0.33	0.01	0.00
x30	0.00	0.00	0.31	0.38	0.31

Control Steps

Figure 10.2 $S_n(i)$ for n = +27, +29 and x30

CDFG Nodes	8	9	10	11	12	13
+27	0.01	0.33	0.66	0.00	0.00	0.00
+29	0.00	0.66	0.33	0.01	0.00	0.00
x30	0.00	0.00	0.31	0.69	0.69	0.31

Control Steps

Figure 10.3 $l_n(i)$ for n = +27, +29 and x30

CDFG Nodes	8	9	10	11	12	13
+27	0.01	0.34	1.00	1.00	1.00	1.00
+29	0.00	0.66	0.99	1.00	1.00	1.00
x30	0.00	0.00	0.00	0.31	0.68	1.00

Control Steps

Figure 10.4 $eb_n(i+1)$ for n = +27, +29 and x30

10.2.4. Fuzzy Schedulability

Given the fuzzy lifetimes, we may calculate the *conflict-free* lifetime of every *allocated* CDFG element as a fuzzy variable. More formally, we define the fuzzy schedulability, $schd_{x,C}$, for a CDFG element x allocated to a cell already allocated to a set C of other CDFG elements, as a fuzzy variable which represents the set containing all control steps in which x will be *live* without resource conflicts. Intuitively,

CDFG Nodes	8	9	10	11	12	13
+27	0.99	0.67	0.00	0.00	0.00	0.00
+29	1.00	0.34	0.01	0.00	0.00	0.00
x30	1.00	1.00	1.00	0.68	0.31	0.00

Control Steps

Figure 10.5 $sa_n(i-d(n)+1)$ for $n = +27, +29$ and x30

CDFG Edges	8	9	10	11	12	13
e27	0.01	0.33	1.00	1.00	1.00	0.84
e29	0.00	0.66	0.99	0.69	0.31	0.00

Control Steps

Figure 10.6 $l_x(i)$ for x = e27 and e29

$schd_{x,C}(i)$ measures the belief that, in the final design, x will be live in control step i and *none* of the CDFG elements in C will be live in i. If $C' \subseteq C$ denotes the subset of C containing all CDFG elements whose lifetimes *may* overlap with that of x (see Section 10.3.3), then we define $schd_{x,C}(i)$ as:

$$schd_{x,C}(i) = Min(live_x(i)\ ,\ \underset{y \in C'}{Min}\ (1 - live_y(i)))$$

That is, $schd_{x,C}$ is calculated as the conjunction of $live_x$ and the complements of $live_y$ for all $y \in C'$. Note that if $C' = \varnothing$ (i.e., if x is allocated to a cell by itself, or if every CDFG element allocated to the same cell as x is either mutually exclusive or has data dependency with x), then $schd_{x,C}(i)$ have maximum values for all i (i.e., $schd_{x,C}(i) = live_x(i)$). This corresponds to the maximally parallel allocation of x, in which there is no possibility of a resource conflict involving x.

10.2.5. Allocation using Fuzzy Schedulability

When allocating a CDFG element, x, with an uncertain lifetime, we may estimate the risk of resource conflicts for each candidate cell based on the fuzzy schedulability $Schd_{x,C}$, where C is the set of CDFG elements already allocated to the candidate cell. Specifically, we define a *schedulability decrement* index, $decr_{x,C}$, as the percentage difference between the peak belief levels for the fuzzy schedulability, $Schd_{x,C}$, and the fuzzy lifetime, $live_x$:

$$decr_{x,C} = \frac{peak_x(live_x) - peak_x(schd_{x,C})}{peak_x(live_x)}$$

where,

$$peak_x(V) = \underset{i=asap(x)}{\overset{alap(x)}{Max}} \; \underset{t=i}{\overset{i+l(x)-1}{Min}} \; V(t)$$

if x is a CDFG node with a latency of $l(x)$, and

$$peak_x(V) = \underset{i}{Max} \; V(i)$$

if x is a CDFG edge.

Intuitively, $peak_x(schd_{x,C})$ measures the maximum belief that the CDFG element x will have a conflict-free lifetime if x is allocated to a cell already allocated to the set C of CDFG elements. If x is a CDFG node, then a conflict-free lifetime requires $l(x)$ consecutive conflict-free control steps, so the peak belief for such a lifetime is calculated as the maximum of minima of $schd_{x,C}(t)$ over a sliding window of $l(x)$ control steps. On the other hand, if x is a CDFG edge, then a conflict-free lifetime may be as short as a single control step, so the peak belief for such a lifetime is simply the maximum of $schd_{x,C}(t)$ for all values of t.

By definition of $Schd_{x,C}$, the value of $peak_x(schd_{x,C})$ cannot exceed the value of $peak_x(live_x)$ (i.e., the belief that a schedule exists for the maximally parallel allocation). Consequently, $decr_{x,C}$ represents a percentage decrease in the maximum belief in the schedulability of x. Alternately, $decr_{x,C}$ may be interpreted as a percentage increase in the *risk* of resource conflicts involving the CDFG element x if it is allocated to a cell already allocated to the set C of CDFG elements. The value of $decr_{x,C}$ ranges from 0 (i.e., $peak_x(live_x) = peak_x(schd_{x,C})$), meaning the risk of resource conflicts is not increased by the allocation, to 1 (i.e., $peak_x(schd_{x,C}) = 0$), meaning inevitable resource conflicts involving x if it is allocated to the same cell as the set C of CDFG elements.

Given the above, we express the increased risk of resource conflicts in terms of hardware costs by multiplying $decr_{x,C}$ by the circuit area required to allocate x to a new cell, $c(op(x))$. Adding this cost factor to the allocation cost functions enables tradeoffs between hardware cost and the risk of resource conflicts during allocation. More formally, if C_{Alloc} denotes the total area of the allocation, then instead of minimizing C_{Alloc}, we modify the cost functions in the allocation algorithm to minimize a new objective function, C'_{Alloc}:

$$C'_{Alloc} = C_{Alloc} + \sum_{x \in CDFG} c_{RC}(x)$$

where $c_{RC}(x)$ is the cost associated with the increased risk of resource conflicts due to the allocation of the CDFG element x. For each CDFG element, x, allocated to a cell already allocated to a set C of other CDFG elements, we define $c_{RC}(x)$ as:

$$c_{RC}(x) = \begin{cases} c_{Penalty} & \text{if } decr_{x,C} = 1 \\ \alpha \times c(op(x)) \times decr_{x,C} & \textit{otherwise} \end{cases}$$

where $c_{Penalty}$ is a high penalty cost for inevitable resource conflicts, and α is a control parameter that defaults to 1.

Basically, the new allocation objective is to avoid inevitable resource conflicts at all costs (due to the high $c_{Penalty}$), and then trade off hardware cost and the risk of resource conflicts based on the control parameter α. For example, if $\alpha = 0$, then the allocation algorithm will try to produce a maximally serial allocation by ignoring the risk of resource conflicts and minimizing the circuit area as much as possible, subject only to inevitable resource conflicts. On the other hand, if α is very large (e.g., 100), then the allocation algorithm will minimize the risk of resource conflicts at the expense of hardware cost, and produce maximally parallel allocations. By adjusting the value of α in a reasonable range (e.g., between 1 and 10), we obtain intermediate allocations with different degrees of tradeoffs between hardware cost and the risk of resource conflicts.

10.3. Implementation Issues

In this section, we discuss a number of implementation issues which arise when extending our SE-based allocation algorithm to accept fuzzy schedules.

10.3.1. Statistical Fuzzy Schedules

The derivation of fuzzy schedules based on dependency analysis has a serious shortcoming in that it only reflects one of the many factors considered by a scheduling algorithm. More importantly, it is computationally expensive, and cannot be readily extended to take advantage of intermixed scheduling and allocation steps. Consequently, we implement a scheme of adapting fuzzy schedules to statistics gathered from previous scheduling iterations.

For each CDFG node n, we keep a running poll, $P_n(i)$ $(i = asap(n),...,alap(n))$, which is a weighted count of the number of times n has been scheduled to control step i in previous scheduling iterations. At the start of synthesis, we set $P_n(i) = 0$ for all values of i. Subsequently, every time n is scheduled in a control step i, we increment $P_n(i)$ by $\frac{1}{Global(n)}$, where $Global(n)$ denotes the global cost of scheduling n in i as calculated by the SE-based scheduling algorithm (Section 5.5). As the scheduling algorithm iterates, the value $P_n(i)$ reflects not only the frequency with which n has been scheduled to different control steps, but also the global costs associated with these control steps. Basically, the higher $P_n(i)$ is, the more confident we can be that n will eventually be scheduled to i. At any point in the synthesis process, we may normalize P_n to obtain an up-to-date fuzzy schedule S_n:

$$S_n(i) = \frac{P_n(i)}{\sum_t P_n(t)}.$$

In essence, we forecast the behavior of the scheduling algorithm based on statistics on the behaviors of past scheduling iterations. This scheme is simple, fast, and reflects all factors considered by the scheduling algorithm without having to analyze the cost functions and heuristics used in the algorithm. More importantly, when scheduling and allocation steps are intermixed, this scheme allows the allocation algorithm to dynamically adapt to changing behaviors of the scheduling algorithm brought on by the allocation of CDFG elements in the first place. In short, it closes the feedback loop between the intermixed scheduling and allocation steps in the synthesis system.

A variation on this scheme is to record in $P_n(i)$ the maximum value of $\frac{1}{Global(n)}$ in previous scheduling iterations. This definition bases the belief of n being scheduled to i entirely on the minimum global cost ever associated with this schedule, and ignores the number of times that n has been scheduled to i. Intuitively, this scheme assigns fuzzy schedules based on the ideal, as opposed to the *typical*, behaviors of the scheduling algorithm. We implement both schemes in the SSE system, and let the users select which scheme to use.

10.3.2. Approximate Schedulability Decrement

The calculation of the schedulability decrement index, $decr_{x,C}$, is computationally expensive because the set, C, of CDFG elements allocated to each cell change frequently as CDFG elements are allocated and de-allocated. To improve the allocation run times, we define a quick approximation, $decr'_{x,C}$, which calculates a lower bound on $decr_{x,C}$ based on the schedulability decrement indices between x and individual elements in C:

$$decr'_{x,C} = \underset{n \in C}{Max}\, decr_{x,\{n\}} \qquad (\leq decr_{x,C}).$$

Consequently, the schedulability decrement between each pair of CDFG elements can be calculated once and cached for subsequent use as long as the underlying fuzzy schedules remain unchanged. This speeds up the allocation iterations by as much as 200% (see Section 10.4). Nevertheless, experiments have shown that, as often as not, allocation using $decr'_{x,C}$ produces a better result than that produced by allocation using $decr_{x,C}$ (see section 10.5).

The formulation of this approximation scheme is inspired by the way [54] calculates the *schedule distance* between two sets of CDFG nodes from the schedule distances between every pair of nodes in either sets.

10.3.3. Dependency Analysis

Dependency analysis yields considerable information on whether different CDFG nodes and edges can have overlapping lifetimes. Basically, two CDFG nodes will never have overlapping lifetimes if one is the descendent of the other (unless they are chained operations), and two edges will never have overlapping lifetimes if the node producing one edge is the descendent of every node which accesses the other edge.

On the other hand, CDFG edges which are either produced or accessed by the same nodes must have overlapping lifetimes.

We detect such impossible and inevitable resource conflicts as follows. At the start of allocation, we traverse the CDFG and collect every pair of dependent nodes. These are checked when calculating the schedulability decrement, $decr_{x,\{y\}}$, between two CDFG nodes or edges x and y. If x and y can never have overlapping lifetimes due to the above dependency analysis, then we set:

$$decr_{x,\{y\}} = decr_{y,\{x\}} = 0.0.$$

On the other hand, if x and y will always have overlapping lifetimes, then we set:

$$decr_{x,\{y\}} = decr_{y,\{x\}} = 1.0.$$

Most of these schedulability decrement values (i.e., all except those for nodes which may be chained together) may be calculated once and then cached as static (i.e., independent of fuzzy schedules) properties of the CDFG. This dependency analysis further reduces the amount of computation for calculating fuzzy schedulability decrements in the course of fuzzy allocation.

10.3.4. Interval Analysis

An additional check for inevitable conflicts must be performed when allocating CDFG nodes due to the non-cumulative property of the fuzzy set operators (i.e., *Max* and *Min*). For example, if three (or more) CDFG nodes must be scheduled in the same two control steps, then they cannot be allocated to the same cell without resource conflicts (unless, of course, they are mutually exclusive). To avoid allocating such CDFG nodes to the same cells, we devise a test for inevitable resource conflicts using interval analysis.

For every CDFG node n with a latency of $l(n)$ control steps, we define the i'th *certainty interval* for n, denoted by $ci(n,i)$ $(i = 1,\dots,l(n))$, as the interval of control steps in which the i'th control step of n's lifetime must lie. For example, if n has a latency of 2, then the first control step in the lifetime of n lies in the interval $[asap(n), alap(n)]$, and the second control step in the lifetime of n lies in the interval $[asap(n)+1, alap(n)+1]$. More formally, we define $ci(n,i)$ as:

$$ci(n,i) = [(asap(n)+i-1), (alap(n)+i-1)] \quad \text{for } i = 1,\ldots,l(n).$$

Consider a set, C, of CDFG nodes which are allocated to the same cell. We may test for inevitable conflicts among nodes in C by first extracting all certainty intervals for all nodes in C, and then searching for an interval of control steps, $[x, y]$, which contains fewer control steps than the number of certainty intervals that are completely enclosed by $[x, y]$. If such an interval exists, then resource conflicts are inevitable when the set C of CDFG nodes are allocated to the same cell.

The actual implementation of this test is facilitated by the incremental nature of our allocation algorithm. In particular, since we allocate one CDFG node at a time, we only need to test for inevitable conflicts involving the node being allocated. For example, when allocating a node n, we check each candidate cell already allocated to a set C of CDFG nodes by extracting all certainty intervals from n and from all nodes in C, and then searching for an interval $[x, y]$ which completely encloses at least one of the certainty intervals of n (i.e., $ci(n,i)$ for some i from 1 to $l(n)$), and which contains fewer control steps than the number of certainty intervals completely enclosed by $[x, y]$. To account for mutually exclusive CDFG nodes, we modify the above test to extract certainty intervals only from those CDFG nodes in C which are *not* mutually exclusive with the node n.

10.4. Experimental Results

The above concepts have been implemented as extensions to our SE-based allocation algorithm. For the EWF example, the speed of fuzzy allocation averages 1.3 CPU second per iteration using the exact schedulability decrement, and 0.6 CPU second per iteration using the approximate schedulability decrement. In terms of final designs, the circuits produced by bottom-up synthesis (i.e., fuzzy allocation, followed by scheduling, followed by final allocation) are comparable to those produced by top-down synthesis (i.e., scheduling followed by allocation) presented in Chapter 8. However, it is not clear whether fuzzy allocation actually helps or impedes the scheduling algorithm.

Consequently, we compare the statistical performance (see Chapter 9) of our SE-based scheduling algorithm with and without fuzzy allocation to determine the effects of fuzzy allocation on our scheduling algorithm. Given a CDFG, we first perform allocation based on fuzzy schedules, and then measure the statistical performance of our scheduling algorithm given this allocation (scheduling an allocated CDFG will be discussed in Section 10.5.3). Basically, if this statistical scheduling performance is better than that obtained with the unallocated CDFG, then fuzzy allocation helps scheduling, otherwise it doesn't. We can also evaluate different fuzzy allocation schemes by comparing the statistical scheduling performance obtained using the allocations produced by these schemes.

For the EWF example with 19 control steps and using pipelined multipliers, we measure the statistical scheduling performance by running a maximum of 100 scheduling iterations for 30 runs, with a target cost of 2 adders, 1 multiplier, 10 registers and 8 interconnects. Table 10.1 summarizes the results of a set of experiments using this benchmark.

Control parameters for the experiments include 3 values of α (0, 1 and 10), the exact and approximate schedulability decrement formulations, and 4 different ways of assigning fuzzy schedules (from the statistical evidence of 100 scheduling iterations, or from dependency analysis with 0, 3 and 6 iterations of successive approximation). For each combination of control parameters, we run 100 iterations of fuzzy allocation, then we record the circuit size for the best allocation found and measure the statistical scheduling performance using this allocation. In Table 10.1, we list the size of each circuit by a string of four numbers, $a-m-r-i$, where a is the number of adders, m is the number of pipelined multipiers, r is the number of registers, and i is the number of 2-input muxes required for interconnects. Moreover, we describe the statistical scheduling performance by first compiling into a *frequency histogram* (see Fig. 10.7)

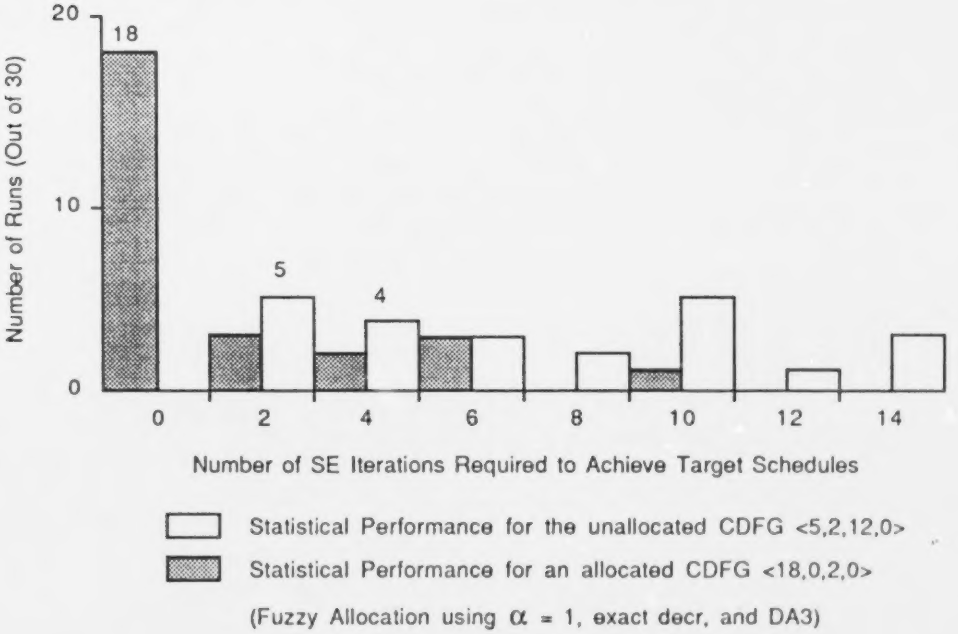


Figure 10.7 Sample Statistical Performance for the EWF

the statistical distribution of the number of scheduling iterations required to achieve the target cost, and then describing this frequency histogram by a tuple of four integers, $<p,n,w,z>$, where p is the peak frequency (i.e., number of runs out of 30) in the histogram, n is the iteration number when p occurs, w is the iteration number by which 20 runs (i.e., two third of 30 runs) have obtained the target cost, and z is the number of runs that never achieve the target scheduling cost in 100 iterations. Fig. 10.7 shows the frequency histogram (described as <5,2,12,0> and <18,0,2,0>, respectively) corresponding to the statistical scheduling performance obtained for the unallocated CDFG and for the fuzzy allocation produced using $\alpha = 1$, exact schedulability decrement, and with fuzzy schedules assigned from dependency analysis with 3 update iterations (i.e., *DA* 3).

S_n	$\text{decr}_{x,C}$	$\alpha = 0$		$\alpha = 1$		$\alpha = 10$	
		Circuit	Statistical Performance	Circuit	Statistical Performance	Circuit	Statistical Performance
stats	exact	2-1-12-23	<8,2,24,0>	2-1-13-20	<30,0,0,0>	4-2-11-24	<3,2,56,4>
	approx	2-1-11-22	<13,2,6,0>	3-1-11-23	<14,2,8,0>	4-2-10-26	<2,56,-,19>
DA0	exact	2-1-9-15	<16,2,8,0>	2-1-9-20	<9,2,12,0>	2-1-13-27	<3,14,56,6>
	approx	2-1-8-19	<7,2,20,0>	2-1-9-18	<17,2,4,0>	2-1-13-29	<7,2,20,0>
DA3	exact	2-1-9-18	<13,2,6,0>	4-1-9-24	<18,0,2,0>	6-1-14-33	<4,10,42,1>
	approx	2-1-9-17	<19,2,4,0>	3-1-8-24	<11,0,12,0>	5-1-13-33	<2,8,-,13>
DA6	exact	3-1-10-23	<13,2,18,0>	4-1-10-26	<9,2,20,0>	5-1-13-36	<4,2,20,0>
	approx	3-1-10-20	<11,2,6,0>	4-1-10-23	<10,2,10,0>	4-1-12-33	<3,2,28,2>

Keys : exact: using exact schedulability decrement
approx: using approximate schedulability decrement
stats: using statistical fuzzy schedules
DA0: using fuzzy schedules from dependency analysis with 0 update
DA3: using fuzzy schedules from dependency analysis with 3 updates
DA6: using fuzzy schedules from dependency analysis with 6 updates

Table 10.1 Experimental Results on the EWF Example (19 Control Steps)

From the results of Table 10.1 (and other similar experimental results), we see that most fuzzy allocations (i.e., all except some produced with $\alpha = 10$) improve the statistical performance of scheduling. Moreover, we make the following observations:

1. The best value of α appears to be 1, and then 0. Several allocations produced with $\alpha = 10$ adversely affects the performance of scheduling. The fact that allocations produced with $\alpha = 0$ (i.e., minimizing hardware cost subject only to inevitable resource conflicts) also improve scheduling performance demonstrates the value of dependency and interval analysis for detecting inevitable resource conflicts.

2. The best fuzzy schedules are assigned by statistical evidence. This is expected since statistical fuzzy schedules are based on actual behavior of the scheduling algorithm. The next best fuzzy schedules are assigned by DA3 (i.e., dependency analysis with 3 updates). Dependency analysis with 6 updates (i.e., DA6) turned out to be worse than simply assuming equal beliefs (i.e., DA0). This shows that an over reliance on dependency analysis is counter-productive since data dependencies may not be the primary factor considered during scheduling.

3. The best fuzzy allocations are produced with exact schedulability decrement. However, the approximate schedulability decrement is just as likely to produce a better fuzzy allocation as the exact schedulability decrement. This can be explained by noting that $decr'_{x,C}$ is a more optimistic formulation than $decr_{x,C}$ (see Section 10.5.2). Evidently this is a reasonable optimism about the ability of the scheduling algorithm.

10.5. Discussions

In this section, we first compare fuzzy allocation with design partitioning (as implemented in [38, 54]). We then discuss alternate formulations for the risk of scheduling conflicts. This is followed by a brief outline of how we extend our SE-based scheduling algorithm to make use of allocation information in a (partially) allocated CDFG.

10.5.1. Comparing Fuzzy Allocation with Design Partitioning

There is a subtle but important difference between fuzzy allocation and design partitioning in terms of how the results of each approach affect subsequent scheduling. In design partitioning, CDFG nodes (and the edges they generate) in each partition are assumed to be in the same subcircuit, which may contain multiple functional units and registers. More importantly, CDFG elements in different partitions are considered to be in separate subcircuits, and will *never* share hardware resources. Consequently, whenever CDFG elements in a partition cannot be scheduled to have non-overlapping lifetimes, the scheduling algorithm assumes that new cells will be added to the corresponding subcircuit, even if cells in other subcircuits may be available. This may increase the size of the design by preventing the scheduling algorithm to consider assigning non-overlapping lifetimes to CDFG elements in different partitions so as to share hardware resources across subcircuits. In contrast, fuzzy allocation does not have this problem since all CDFG elements are considered to be in the same "subcircuit".

10.5.2. Optimistic and Pessimistic Formulations

Due to the use of maximum and minimum functions in fuzzy set operators, one must take great care in the formulation of fuzzy measures. For example, an intuitive formulation for the risk of scheduling conflicts involving CDFG elements n and m may be:

$$conflict(n,m) = \underset{t}{Max}\; Min(live_n(t)\,,\; live_m(t)).$$

That is, the risk of conflict is the maximum belief that n and m will both be live in some control step t. However, this is a pessimistic formulation, which basically says that we believe the scheduling algorithm will assign overlapping lifetimes to CDFG elements n and m *whenever possible*, even if we allocate n and m to the same cell. This produces maximally parallel allocations which degrade the performance of scheduling in basically the same way that allocations produced by $\alpha = 10$ do.

In contrast, the schedulability decrement between n and m, $decr_{n,\{m\}}$, is an optimistic formulation which basically says that we believe the scheduling algorithm will assign to n a lifetime which will not overlap with the lifetime of m whenever possible, if we allocate n and m to the same cell. By the same logic, the approximate schedulability decrement, $decr'_{n,\{m\}} \leq decr_{n,\{m\}}$, is even more optimistic about the ability of the scheduling algorithm to assign non-overlapping lifetimes to CDFG elements allocated to the same cells.

10.5.3. Scheduling with Partial Allocation

Since bottom-up synthesis requires that we perform scheduling after allocation, we must extend our scheduling algorithm to take into account partial allocations in a CDFG. We have implemented two approaches for scheduling a partially allocated CDFG.

In the first approach, we treat the allocations as imposing additional constraints on the scheduling task, in that CDFG elements allocated to the same cells must not be scheduled to have overlapping lifetimes (unless they are mutually exclusive CDFG elements). These *allocation constraints* are treated as *soft* constraints in the sense that schedules which violate allocation constraints are still considered valid schedules. However, we bias the scheduling algorithm against violating these constraints by adding a penalty cost component to the scheduling cost functions. More formally, if C_{Schd} denotes the overall hardware cost of a schedule, then instead of minimizing C_{Schd}, we modify the cost functions in the scheduling algorithm to minimize a new objective function, C'_{Schd}:

$$C'_{Schd} = C_{Schd} + \sum_{x \in CDFG} c_{AC}(x)$$

where $c_{AC}(x)$ is the cost for violating the allocation constraint (if any) of the CDFG element x. For every CDFG element x, we define $c_{AC}(x)$ as:

$$c_{AC}(x) = \begin{cases} 0 & \text{if } allocation\ constraint\ of\ x\ is\ satisfied \\ c(op(x)) & otherwise \end{cases}$$

where $c(op(x))$ is the circuit area required for allocating x to a new cell. To check for violation of allocation constraints, we traverse the tally data structure, and check each set $tally(op, t)$ for members which are allocated to the same cells but are not mutually exclusive.

This implementation works very well if a conflict-free schedule exists for the allocated CDFG. For example, if we perform design iterations in a top-down manner (i.e., scheduling and then allocation, and then iterate), then this approach finds good conflict-free schedules very quickly since conflicting allocations serve to steer the scheduling algorithm from blind alleys in the solution space. On the other hand, if a conflict-free schedule does not exist for the allocated CDFG, as in the case of allocation based on fuzzy schedules, then this implementation often fails to find a good schedule because it tries to reduce allocation conflicts at the expense of hardware costs.

In the second approach, we treat the allocations as *hints*, rather than constraints, to the scheduling task. This is implemented in the greedy scheduling step in our SE-based scheduling algorithm. Previously, the candidate schedules for each CDFG node consist of all control steps from the as soon as possible (ASAP) to the as late as possible (ALAP) schedules for the node. Now, for an allocated node, we redefine its candidate schedules to exclude those control steps which will lead to overlapping lifetimes between the node and other CDFG nodes already allocated to the same cell. If this results in a CDFG node having no candidate schedules, then a resource conflict is inevitable, so we revert the definition of candidate schedules to all control steps between ASAP and ALAP schedules. However, we do not consider allocation conflicts in either the global cost function or the objective function (i.e., we still minimize C_{Schd}). Basically, this approach tries to schedule CDFG nodes to avoid allocation conflicts whenever possible, but does not penalize schedules containing such conflicts. We have found through experimentation that this implementation works much better for bottom-up synthesis (i.e., allocation and then scheduling) than the first approach, perhaps due to its greater tolerance for inevitable resource conflicts. The experimental results in Table 10.1 are obtained using this second approach.

10.6. Summary

In this chapter, we have presented a new formulation of the allocation task for fine-grained bottom-up synthesis. Basically, we viewed unscheduled operations as having uncertain schedules, and applied fuzzy set theory to quantify these uncertainties during allocation. This allowed the allocation algorithm to trade off hardware area and the risk of resource conflicts. Consequently, allocation is generalized to the task of finding the smallest circuit for which a conflict-free schedule is still highly possible. We have described the implementation of fuzzy allocation as extensions to our SE-based allocation algorithm, and presented experimental results which demonstrate that fuzzy allocations improve the statistical performance of the scheduling algorithm. Finally, we have compared fuzzy allocation with design partitioning, discussed optimistic and pessimistic formulations of fuzzy schedules, and presented two schemes for scheduling a (partially) allocated CDFG.

Chapter 11. Integrating Scheduling and Allocation

Given the work presented in previous chapters, we are now ready to describe the integration of scheduling and allocation tasks in the SSE system. As proposed in Chapter 2, we organize the SE-based scheduling and allocation algorithms as two concurrent, algorithmic agents in a blackboard architecture as depicted in Fig. 11.1.

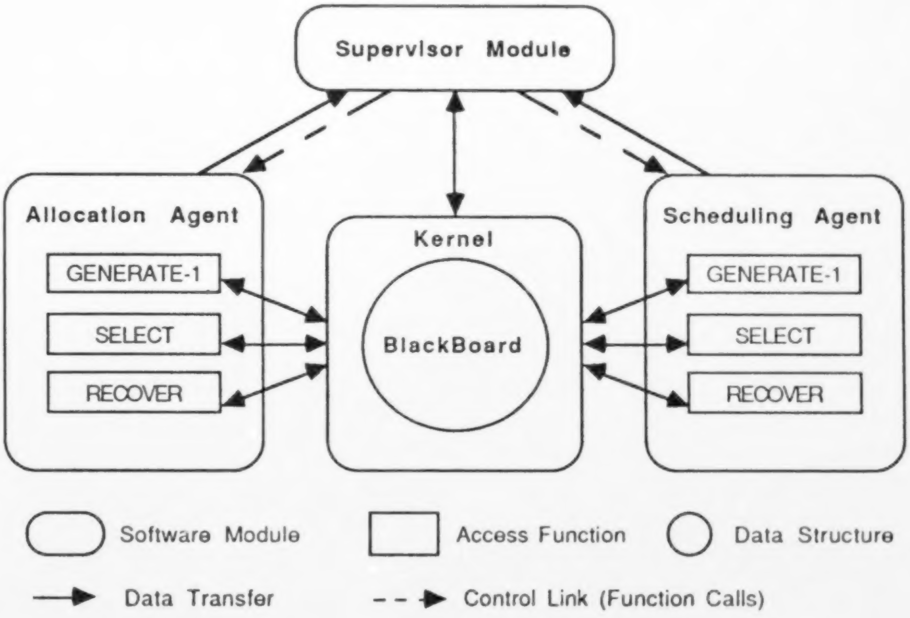

Supervisor Module
Allocation Agent
GENERATE-1
SELECT
RECOVER
Kernel
BlackBoard
Scheduling Agent
GENERATE-1
SELECT
RECOVER
Software Module
Access Function
Data Structure
Data Transfer
Control Link (Function Calls)

Figure 11.1 The Blackboard/Agent Architecture in SSE

In the remainder of this chapter, we will first describe the Blackboard and kernel software, and discuss the implementation of algorithmic agents in SSE. We will then present two control mechanisms by which the supervisor module can coordinate scheduling and allocation steps and implement different synthesis strategies in SSE. Finally, we will present experimental results for a number of synthesis strategies to demonstrate the flexibility of this framework for integrating scheduling and allocation.

11.1. Blackboard and Kernel Software

At the center of SSE is a centralized data structure, namely the Blackboard, and the kernel software which provides access to all data posted on the Blackboard. The primary data on the Blackboard are the CDFG, the CG (i.e., the data path circuit graph), the schedules (which specify the control path state graph), and the allocations (from which we may derive the control path circuitry). Other data on the Blackboard include control parameters, user-specified initializations and/or constraints, and status variables via which agents communicate with each other and with the supervisor module.

The kernel software serves two important functions. First, it implements data independence between agents and the Blackboard by translating between the data representation of the Blackboard and that of the individual agents whenever necessary. For example, whenever the allocation algorithm decides to permute inputs of a CDFG node to make better use of existing connections in the data path circuit (see Section 7.8), instead of actually changing the CDFG (which might have caused serious problems to the scheduling algorithm), we record the permutation in the Blackboard. All access functions for the allocation algorithm take into account such permutations so that, as far as the the allocation algorithm is concerned, it appears as if the CDFG had been modified as requested.

The second function implemented by the kernel software is that of maintaining data consistency in the Blackboard via lazy evaluation (i.e., updating data whenever they are accessed rather than when they first become out of date). For example, whenever a node is scheduled or de-scheduled, the fuzzy lifetimes for its dependent nodes and edges may become out of date. Since there may be other scheduling/de-scheduling steps before the next allocation/de-allocation step, we defer updating the fuzzy lifetimes until the next time they (or other fuzzy measures derived from them) are accessed so as to reduce unnecessary computation. This lazy update is implemented in the kernel access functions: whenever schedules are changed by the scheduling algorithm, the appropriate fuzzy lifetimes are invalidated; and whenever fuzzy measures are accessed by the allocation algorithm, all invalidated fuzzy lifetimes are updated before the access functions retrieve any data.

11.2. Algorithmic Agents

In SSE, scheduling and allocation are performed incrementally by algorithmic agents. Conceptually, an algorithmic agent is an autonomous software object which continually performs a particular task (e.g., SE-based scheduling or allocation). We implement such an algorithmic agent as a software module which has its own internal states (i.e., persistent, private variables) and which provides three access functions: *GENERATE*−1, *SELECT* and *RESTORE* (see Fig 11.1). Each of these functions performs a small step in SE-based synthesis. Given a partial solution, the function *GENERATE*−1 picks one unassigned variable with the maximum *PRIORITY* and assigns to it a candidate value with the minimum *INCR* cost (i.e., the inner loop of *GENERATE*). If this produces a complete solution (i.e., all variables have been assigned values), then *GENERATE*−1 also records this new solution if it is the best found so far. The function *SELECT* is the same function in SE which probabilistically discards inferior elements of a complete solution and produces a partial solution. Finally, the function *RESTORE* simply reinstates the best solution found as the current solution.

Given the above, the supervisor module can implement SE-based scheduling (or allocation) in SSE by issuing the following function calls. First, the supervisor repeatedly calls *GENERATE*−1 in the scheduling (allocation) agent until a complete schedule (allocation) is obtained. Second, the supervisor calls *SELECT* to generate a partial schedule (allocation). Third, the supervisor iterates between the above two steps until the termination criteria are met. Finally, the supervisor calls *RESTORE* in the scheduling (allocation) agent to restore the best schedule (allocation) as the current schedule (allocation).

Consequently, we can easily implement either purely top-down (i.e., SE-based scheduling followed by SE-based allocation) or purely bottom-up (i.e., SE-based allocation followed by SE-based scheduling) synthesis with iterations. More importantly, since our scheduling and allocation algorithms can handle partial allocation and incomplete schedules, the supervisor module can intermix scheduling and allocation steps (i.e., scheduling, de-scheduling, allocation, and de-allocation steps) in arbitrary order. This permits more sophisticated synthesis strategies than purely top-down and purely bottom-up synthesis.

11.3. Control Mechanisms

We implement two control mechanisms by which the supervisor module can coordinate the scheduling and allocation steps in SSE: intermixing of synthesis steps, and focusing of attention for individual agents.

11.3.1. Intermixing of Synthesis Steps

In SSE, the supervisor can dynamically decide whether greedy scheduling or greedy allocation (of a single CDFG node or edge) should proceed next; and once all CDFG nodes (and edges) are scheduled (allocated), the supervisor can dynamically decide whether probabilistic de-scheduling (de-allocation) should proceed. Then the system will iterate by re-scheduling (re-allocating) all de-scheduled (de-allocated) CDFG nodes (and edges). This allows us to experiment with different synthesis strategies just by changing the supervisor module in SSE.

For example, given a partially scheduled and partially allocated CDFG, the supervisor module may implement a "highest priority first" policy by scheduling a node whenever the priority for scheduling this node is higher than that for allocating a CDFG element, and vice versa. However, rather than directly comparing the *PRIORITY* values between scheduling and allocation, we compare a *priority index* between scheduling and allocation. We define the priority index for scheduling (or allocation) as the ratio, $\frac{f}{s}$, where f is the highest *PRIORITY* value, and s is the second highest *PRIORITY* value, for all unscheduled (unallocated) CDFG nodes and edges. This scheme gives preference to scheduling/allocating the node/edge with the highest *relative PRIORITY* values.

Alternately, the supervisor module may implement a "minimum cost first" policy by scheduling a node whenever the incremental cost for scheduling this node is lower than that for allocating a CDFG element, and vice versa. Again, rather than directly comparing the *INCR* costs between scheduling and allocation, we compare a *incremental cost index* between scheduling and allocation. We define the incremental cost index for scheduling (or allocation) as the ratio, $\frac{f}{s}$, where f is the lowest *INCR* cost, and s is the second lowest *INCR* cost, for all candidates of the next node or edge to be scheduled (allocated). This schemes gives preference to scheduling/allocating the node/edge with the lowest *relative INCR* costs.

On the other hand, given a completely scheduled and completely allocated CDFG, the supervisor module may implement a "fine-grained integration" policy by always de-scheduling and de-allocating the CDFG as soon as possible. Under this scheme, SSE always performs scheduling with a partial allocation and allocation with an incomplete schedule. Alternately, the supervisor module may implement other, coarser-grained integration schemes, in which either de-scheduling or de-allocation may be selectively delayed so as to incorporate scheduling with complete allocations, or allocation with complete schedules, or both. For example, the supervisor may implement a "primarily top-down" policy by performing 1 de-scheduling step per 50 de-allocation steps (and after restoring the best allocation found), or a "primarily bottom-up" policy by performing 1 de-allocation step per 50 de-scheduling steps (again, after restoring the best schedule found).

11.3.2. Focusing of Attention

In SSE, the supervisor module can also dynamically restrict the CDFG nodes and edges which are subject to change by the scheduling or the allocation agents. Specifically, we define a *focus* for a scheduling (or allocation) agent as the set of CDFG nodes (and edges) which can be scheduled (allocated) or de-scheduled (de-allocated) by the agent. Intuitively, this focuses the attention of the synthesis agents to particular subsets of the CDFG, since each agent will try to optimize a design by ripping-out and reconstructing only elements in its focal subset of the CDFG. This control mechanism facilitates the implementation of even more complex synthesis strategies.

For example, the supervisor module may implement a "expanding focus" policy by initially focusing the scheduling agent on the critical path nodes and the allocation agent on the non-critical path nodes and edges. Subsequently, the supervisor module expands the focus of each synthesis agent to include more and more CDFG elements, until eventually the entire CDFG is under focus. Basically, this scheme tries to establish, early on, a good critical path schedule and a good allocation for non-critical path CDFG elements, in the hope that subsequent synthesis steps will be positively influenced by these initial, partial designs.

Alternately, the supervisor module may implement a "contracting focus" policy by initially focusing the synthesis agents on the entire CDFG, and then gradually reducing the scheduling focus to CDFG nodes with high scheduling costs, and reducing the allocation focus to CDFG elements with high allocation costs. A simple implementation of this scheme is to use the normalized global costs in SE as the scheduling/allocation costs for individual CDFG elements. A more complex implementation may use statistical evidence similar to those presented in Section 10.3.1 to detect CDFG elements whose schedules (allocations) are *probably optimum*, and then exclude these elements from the scheduling (allocation) focus. Intuitively, this scheme presumes that relatively few elements in the CDFG are responsible for poor scheduling and/or poor allocation, thus the synthesis agents should try to identify such CDFG elements and then concentrate on these to optimize the design.

On the other hand, the supervisor module may incorporate higher level design knowledge (than the synthesis agents) and use the focusing mechanism to implement hierarchical or windowing techniques [23], or impose design partitioning (see Section 10.5.1) on the synthesis agents. In all cases, a large CDFG is divided into a number of possibly overlapping subgraphs, which are separately synthesized (i.e., scheduled and allocated) and then resynthesized as an entire CDFG. In hierarchical synthesis, the CDFG is divided along the design hierarchy; in windowed synthesis, the CDFG is divided by neighborhood size (e.g., scheduling a certain number of connected CDFG nodes at a time); in design partitioning, the CDFG is divided according to physical design considerations. Basically, these techniques apply the principle of "divide and conquer" to try and reduce the computational complexity of scheduling/allocating a large CDFG.

11.4. Experiments with Synthesis Strategies

In this section, we present a number of synthesis strategies which we have experimented with in the SSE system. Note that our intention is not to develop a definitive synthesis strategy, but to demonstrate the flexibility and strength of our integration framework for high level synthesis.

11.4.1. Fine-Grained Integration

The first synthesis strategies we tried are fine-grained integration schemes, in which de-scheduling is performed whenever all CDFG nodes are scheduled, and de-allocation is performed whenever all CDFG nodes and edges are allocated. We experimented with three policies for intermixing greedy scheduling and greedy allocation steps: namely "random" (i.e., randomly intermixing scheduling and allocation steps with user-specified probability distributions), "highest priority first", and "minimum cost first" policies. Unfortunately, all of these schemes have very poor performance in that they require large numbers of iterations to generate good designs. This is understandable when one considers that fine-grained integration has a much larger design space than that for either the scheduling or allocation task alone. This leads us to conclude that a coarser-grained integration strategy is necessary to impose certain structures to the synthesis process to reduce the design space and thereby improve synthesis performance.

11.4.2. Top-Down and Bottom-Up Synthesis With Iterations

The most straightforward coarse-grained integration strategies are purely top-down and purely bottom-up synthesis with iterations. We implement top-down (or bottom-up) synthesis by performing a fixed number of SE-based scheduling (allocation) iterations, restoring the best schedule (allocation) found, and then performing a fixed number of SE-based allocation (scheduling) iterations, again restoring the best allocation (schedule) found. This process is then repeated a number of times so that scheduling is guided by the best allocation found in previous allocation runs, and allocation is guided by the best schedule found in previous scheduling runs. While these strategies produced good designs in reasonable run times, we were not able to achieve better designs than those generated by separate SE-based scheduling and allocation runs (as presented in Chapter 8).

11.4.3. Allocation based on Iterative Improvement

The next strategy we tried is to maintain the same schedule while repeatedly performing a fixed number of SE-based allocation (i.e., invoking *RESTORE* in the allocation agent after producing every n complete allocations). This implements a new allocation algorithm based on iterative improvement as shown in Fig. 11.2.

This algorithm generates a new solution from the current solution by performing n iterations of SE-based allocation. The new solution is then accepted as the current solution if and only if it is better than the current solution. This is repeated for a total of m times, requiring $m \times n$ SE iterations. This iteratively improves the current solution by performing a probabilistic search over the neighborhood of the current solutions. In fact, we may consider the straightforward SE algorithm as a special case of the above algorithm (i.e., with $m = 1$).

```
Algorithm Allocate(m,n);
 /* repeat m times of n iterations of SE-based Allocation */
 begin
   X := GENERATE(∅);
   $X_{best}$ := X;
   loop for m times;
     begin
       X := $X_{best}$;
       loop for n times;
         begin
           p := SELECT(X);
           X := GENERATE(p);
           if (C(X) < C($X_{best}$)) then $X_{best}$ := X;
         end;
     end;
   return($X_{best}$);
 end Allocate;
```

Figure 11.2 Pseudo-Code for Iterative Improvement Based Allocation

Fig. 11.3 plots the distribution of the best circuit costs obtained for the EWF example (with 19 control steps and pipelined multipliers) over 10 allocation runs for different combinations of m and n values. Note that in each case, a total of 3000 SE iterations are performed. The case of $m = 1$, $n = 3000$ corresponds to performing 3000 iterations of the normal SE-based allocation algorithm; the case of $m = 3000$, $n = 1$ corresponds to performing 3000 iterations of greedy search (i.e., a new solution is accepted as the current solution if and only if the new solution has a lower cost), where each new solution is generated from the current solution, X, by applying *SELECT* to X and then applying *GENERATE* to the resulting partial solution. The other cases correspond to performing m iterations of greedy search, except each new solution is generated by performing n SE iterations starting from the current solution and then restoring the best solution found. (To put the circuit costs in Fig. 11.3 in perspective, HAL's circuit for the same example in Fig. 8.7(a) has a cost of 280.5, and SSE's circuit in Fig. 8.7(b) has a cost of 246.5.)

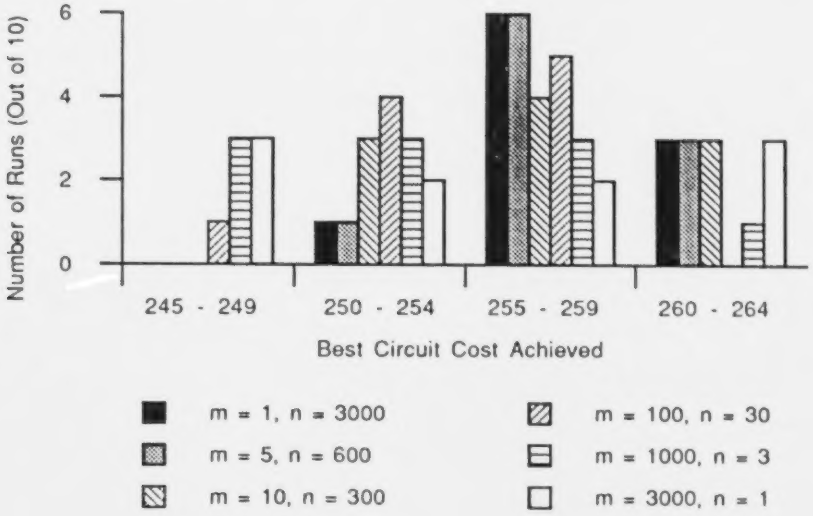


Figure 11.3 Allocation Performance for Select Values of *m* and *n*

These and similar experimental results suggest that it may be better to perform many runs (e.g., 1000) of SE-based allocation, starting from successively better solutions and performing a few SE iteration (e.g., 3) in each run, than to perform a single run of SE-based allocation for a large number of SE iterations. However, while the total number of SE iterations remain the same, using larger *m* values increases the run time for the algorithm due to the overhead in restoring the best solutions for *m* times.

Since performing small numbers of SE iterations starting from successively better solutions can be viewed as searching over the *close by* neighborhoods of such solutions, we suspect that the design space for allocation may be such that near optimal designs are clustered close together. This makes intuitive sense because most of the near-optimal circuits differ only in a few multiplexers, and hence should have largely similar allocations. This observation leads us to formulate the goal-directed synthesis strategy in the next subsection.

11.4.4. Goal Directed Synthesis

In goal directed synthesis, the idea is to optimize the critical paths with respect to time (i.e., scheduling before allocation), and optimize the non-critical paths with respect to hardware (i.e., allocation before scheduling). This approach is commonly used by IC designers, and has been applied by CAMAD [49] to formulate a multi-level optimization strategy which selects different design transformations depending on whether the nodes to be changed lie on the critical paths or not. However, unlike CAMAD, we are able to change the focus of our scheduling and allocation agents to limit our search to the most promising regions of the design space. The following steps apply the above ideas in SSE:

1. Perform 100 iterations of SE-based scheduling with the focus on all CDFG nodes on the most critical paths (i.e., nodes whose freedoms are 1 if the global maximum timing constraint is set to the critical path length of the CDFG), and then restore the best partial schedule found.
2. Perform 300 iterations of SE-based allocation on the entire CDFG based on the best partial schedule and based on statistical fuzzy schedules for all unscheduled nodes, and then restore the best allocation found.
3. Perform 100 iterations of SE-based scheduling with the focus on all CDFG nodes on the secondary critical paths (e.g., nodes whose freedoms are 2 if the global maximum timing constraint is set to the critical path length of the CDFG), and then restore the best partial schedule found.
4. Perform 300 iterations of SE-based allocation on the entire CDFG based on the best partial schedule and based on statistical fuzzy schedules for all unscheduled nodes, and then restore the best allocation found.
5. Perform 100 iterations of SE-based scheduling with the focus on all

unscheduled CDFG nodes (in this case, nodes whose freedoms are greater than 2 if the global maximum timing constraint is set to the critical path length of the CDFG), and then restore the best complete schedule.

6. Perform 300 iterations of SE-based allocation on the entire CDFG based on the best complete schedule, and then restore the best allocation found.
7. Repeat Step 6. for a total of 10 times, each time reducing the allocation focus by removing 1 CDFG node or edge with the minimum *GLOBAL* cost.

For the EWF example, with a maximum timing constraint of 19 control steps and using pipelined multipliers, the above synthesis steps generated a circuit containing 2 adders, 1 multiplier, 11 registers and 15 equivalent 2-to-1 multiplexers in 1921 CPU seconds (or 32 CPU minutes). Table 11.1 shows the schedule and allocation for this design, and Fig. 11.4 shows the corresponding circuit. Note that this is a smaller circuit than our previous best design, which contains 16 (instead of 15) equivalent 2-to-1 multiplexers (see Fig. 8.7). While this by itself does not mean that goal directed synthesis will always produce better designs, the fact that it actually found a better design than purely top-down/bottom-up synthesis does lend support to our contention that intermixing scheduling and allocation steps *may* produce better designs.

To gain further insight into the above synthesis steps, we perform the following three experiments. In the first experiment, we replace steps 1 to 5 by straightforward SE-based scheduling, and then perform steps 6 and 7. After 10 such synthesis runs, the best circuit generated contains 2 adders, 1 multiplier, 11 registers and 18 equivalent 2-to-1 multiplexers. This suggests that the above optimum design is not a result of steps 6 and 7 being a particularly good allocation strategy. In the second experiment, we initialize the EWF CDFG with the schedule in Table 11.1, and then perform 10 runs of allocation using steps 6 and 7 (and starting from the unallocated CDFG). The best circuit generated by these runs contains 2 adders, 1 multiplier, 11 registers and 17

	ADD1	ADD2	MULT	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11
IN				E33	E38	E18	E13	E26	E1			E39	E2	
1	+3	+32		E32	E38	E18	E13	E26	E1			E39		E3
2		+12		E32	E38	E18		E26	E1	E12		E39		E3
3	+20			E32	E38	E18	E20		E1	E12		E39		E3
4		+25		E32	E38	E18			E1	E12		E39	E25	E3
5			x21	E32	E38	E18			E1	E12		E39	E25	E3
6				E32	E38	E18			E1	E12	E21	E39	E25	E3
7		+19	x24	E32	E38	E18	E19		E1	E12		E39	E25	E3
8	+22	+11		E32	E38	E18	E19	E22	E1		E24	E39	E11	E3
9		+27	x9	E32	E38	E18	E19	E22	E1		E27	E39		E3
10		+29		E9	E38	E18	E19	E22	E1		E27	E39	E29	E3
11	+23	+8	x30		E38	E18	E19	E26	E1	E8	E27	E39		E3
12	+10	+7		E30	E38	E18	E10	E26	E1	E8	E27	E39	E7	
13	+15	+31	x6	E31	E38	E18	E15	E26	E1	E8	E27	E39		
14	+28	+41	x16	E31	E38	E18		E26	E1	E8	E6		E41	E28
15	+4	+35		E31	E38	E18		E26		E8	E16	E4	E41	E35
16	+17		x36	E31	E38	E18		E26		E8		E4	E41	E35
17	+14	+5	x40	E31	E38	E18	E13	E26			E36		E5	E35
18		+37		E31	E38	E18	E13	E26			E43		E5	E35
19	+42	+34		E33	E38	E18	E13	E26			E43	E39	E5	

Table 11.1 Best Schedule and Allocation for the EWF Example

equivalent 2-to-1 multiplexers. This suggests that, while it may be easier to generate a good allocation from the schedule in Table 11.1 than from other (equally *optimum*) schedules, the initial allocation produced by step 4 is very important to the performance of goal directed synthesis. In the third experiment, we modify step 7 so that it does not reduce the allocation focus at all, and perform 10 runs of this version of goal directed synthesis. The best circuit generated by these runs contains 2 adders, 1 multiplier, 10 registers and 20 equivalent 2-to-1 multiplexers. This suggests that contracting

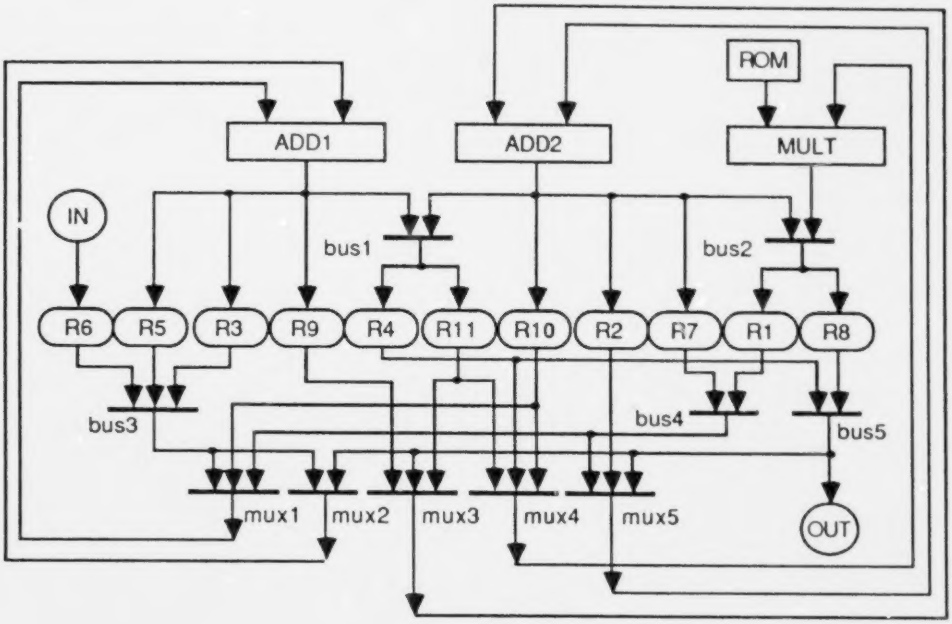

ROM
ADD1
ADD2
MULT
IN
bus1
bus2
R6
R5
R3
R9
R4
R11
R10
R2
R7
R1
R8
bus3
bus4
bus5
mux1
mux2
mux3
mux4
mux5
OUT

Figure 11.4 Best Circuit for the EWF Example

allocation focus does improve the performance in the allocation steps.

Given the above, we conclude that intermixing scheduling and allocation steps can guide the scheduling algorithm towards a schedule which may result in better allocations than other schedules with equal costs (i.e., C_{Schd}). Moreover, the allocation produced on the basis of fuzzy schedules can serve to establish a good starting point for the final allocation steps. Finally, a policy of contracting focus can improve the performance of allocation.

11.5. Summary

In this chapter, we have described the integration of scheduling and allocation algorithms in the SSE synthesis system. Basically, we organize the SE-based scheduling and allocation algorithms as concurrent algorithmic agents around a blackboard architecture. We have described the Blackboard and its associated kernel software, the implementation of algorithmic agents, and the two control mechanisms by which the supervisor module coordinates different synthesis agents.

To demonstrate the flexibility of this integration framework, we have presented a number of synthesis strategies that we have experimented with. Through such experimentation, we have found that fine-grained integration of scheduling and allocation is ineffective because the resulting design space is too large. We have also found that iterative improvement based on SE-based allocation consistently achieves better designs than the straightforward SE-based allocation algorithm. This has led us to formulate a goal-directed synthesis strategy, in which we focus the scheduling algorithm on the critical paths of a CDFG, allocate the entire CDFG, and then schedule the non-critical paths of the CDFG on the basis of this allocation. Experimental results have shown that this synthesis strategy can produce better designs than either purely top-down or purely bottom-up synthesis with iterations. This clearly demonstrates the advantage of being able to arbitrarily intermix scheduling and allocation steps in the SSE system.

Chapter 12. Conclusion

In this chapter, we will summarize this thesis, reiterate the major contributions of this work, and then conclude with suggestions for directions for future work.

12.1. Summary

In this thesis, we have proposed a framework in which different tasks in high level synthesis are solved concurrently by algorithmic agents organized in a blackboard architecture. Such a framework will allow a tight coupling between different synthesis tasks, and will facilitate experimenting with different synthesis strategies. However, in order to implement this synthesis system, we have to address two issues. First, we need new synthesis algorithms which are fast, incremental, and produce good designs. Second, we need new bottom-up synthesis techniques in order to perform low level synthesis tasks before higher level tasks are completed. In this thesis, we have addressed these issues by focusing on integrating the tasks of scheduling and allocation in the prototype SSE system.

For fast, incremental and effective synthesis algorithms, we have applied the technique of Simulated Evolution (or SE) to the tasks of scheduling and allocation. We have first introduced SE as a general optimization technique based on the analogy between optimization and evolutionary processes, and then described our implementation of SE in the context of an *optimum assignment* problem. This separates all application specific components of SE into three cost functions: *PRIORITY*, *INCR* and *GLOBAL*. We then described our application of SE to the tasks of scheduling and allocation by first formulating each of these tasks as an optimum assignment problem, and then defining the cost functions *PRIORITY*, *INCR* and *GLOBAL* for each task. We have also outlined a number of important extensions to the basic SE-based scheduling and allocation algorithms, to show that such extensions are easily implemented by changing the cost functions in these algorithms.

We have presented experimental results of our SE-based scheduling and allocation algorithms on a number of design examples from the literature. These results have shown that, compared to other synthesis systems, our SE-based algorithms generate comparable designs very quickly, and generate much better designs when given longer run times. In particular, our designs for the EWF benchmark are the best designs ever reported to date on this popular benchmark. Moreover, we have compared SE and Simulated Annealing (or SA) using the scheduling problem. Experimental results have shown that SE may be better than many implementations of SA because it implements a more directed search than SA, and because it is capable of more distant jumps in the solution space than most implementations of SA. By combining SE and SA into a single algorithm, we have also demonstrated that SE does not benefit from a "cooling schedule" as does SA.

For bottom-up synthesis, we have presented a new formulation of the allocation problem based on the concept of fuzzy schedules. Basically, we view unscheduled nodes as having uncertain schedules, and apply fuzzy set theory to represent our beliefs that some control steps are better scheduling candidates than others. From such fuzzy schedules, we derived the concepts of fuzzy lifetime and fuzzy schedulability, and then defined the fuzzy schedulability decrement metric which measures the incremental risk of scheduling conflicts due to each candidate allocation. We have described the implementation of these concepts as simple extensions to our SE-based allocation algorithm, and we have presented experimental results which demonstrate the effectiveness of this approach for bottom-up synthesis.

Finally, we have described the integration of the above SE-based scheduling and allocation algorithms in the SSE system. We have described two control mechanisms by which the supervisor module in SSE can coordinate the scheduling and allocation steps and implement different synthesis strategies. To demonstrate the flexibility of this integration framework, we have presented a number of synthesis strategies that we have experimented with. In particular, we have found that iterative improvement based on SE-based allocation steps consistently achieves better performance than the basic SE-based allocation algorithm. This has led to a goal-directed synthesis strategy in which we focus the scheduling algorithm on the critical paths of the CDFG, allocate the entire CDFG, and then schedule the non-critical paths on the basis of this allocation. Experimental results have shown that this ad hoc strategy can produce better designs than either purely top-down or purely bottom-up synthesis with iterations. This clearly demonstrates the advantage of the integration framework in SSE.

12.2. Major Contributions

There are three major contributions in the work presented in this thesis. The first contribution is to demonstrate the value of effective design space exploration in scheduling and allocation. By applying simulated evolution to scheduling and allocation, we have shown that simple heuristics and cost functions can produce good designs very quickly if the CPU times are devoted to effective exploration of the design space instead. In particular, we have shown that the performance of SE-based synthesis algorithms is primarily due to their ability to find locally optimal solutions in a solution subspace, as well as their ability to rapidly search over large regions of the design space.

The second contribution of this work is to generalize the allocation task to allow for full allocation of all CDFG nodes and edges without scheduling. By applying fuzzy set theory to represent and reason about uncertain lifetimes of CDFG nodes and edges, we have shown that allocation based on fuzzy schedules can improve the performance of subsequent scheduling runs. More importantly, we have demonstrated (with goal-directed synthesis) that, while different schedules may have the same "optimum" costs, they are not necessarily equal in terms of the allocation algorithm's ability to generate a good circuit from these schedules. Bottom-up synthesis has the advantage of being able to guide the scheduling algorithm towards those schedules which are more suited to the allocation algorithm.

The third contribution of this work is to demonstrate the feasibility of our integration framework for synthesis tasks. We have shown how SE-based scheduling and allocation algorithms can be easily incorporated as autonomous, algorithmic agents. We have also presented experimental synthesis strategies to demonstrate the flexibility of our integration framework. The advantage of the SSE system is evident in the fact that we were able to produce a better design with goal-directed synthesis than with purely top-down or purely bottom-up synthesis.

12.3. Suggestions for Future Work

There are three major directions for future research. The first direction for future work is to integrate other synthesis tasks using the same principles. For example, to incorporate control path generation into SSE, we would first apply simulated evolution to the task of control path optimization, and then generalize the task of control path generation to accommodate fuzzy schedules and *fuzzy allocations* (i.e., uncertainties in the allocation of CDFG elements to hardware cells). On the other hand, in order to incorporate functional optimization into SSE, we would apply simulated evolution to the task of functional optimization, and then generalize the task of scheduling to accommodate *fuzzy CDFG*'s (i.e., uncertainties in the CDFG itself).

The second direction for future work is to investigate more sophisticated synthesis strategies. For example, a more sophisticated goal-directed synthesis strategy may focus the synthesis agents based on additional factors (than critical paths) such as allocation criticality, probable/possible cost improvements, specialized heuristics, etc. Another approach may define a suite of "expert" synthesis strategies in different supervisor modules, and implement a meta-supervisor which selects the appropriate supervisor module to use based on characteristics of the CDFG, user hints, and/or previous synthesis results on similar CDFG's.

The third direction for future work is to study variations of the SE algorithm as general probabilistic search algorithms. For example, iterative improvement based on SE (see Fig. 11.2) may be a good probabilistic search algorithm in itself, especially for applications such as allocation where good solutions seem to be close to one another. Other examples of variant SE algorithms worth studying arise when different dynamic focus strategies are used. In large solution spaces, it may be necessary to structure the probabilistic search of SE with a "focusing schedule" (analogous to the cooling schedules in simulated annealing) which dynamically changes the focus in a problem independent manner. This will control the evolution process by adjusting the maximum state transition distance, and by selecting the solution subspaces to be examined.

References

1. E. H. L. Aarts and P. J. M. van Laarhoven, "Statistical Cooling: A General Approach to Combinatorial Optimization Problems," *Philips Journal of Research*, vol. 40, pp. 193-226, 1985.
2. G. Borriello and E. Detjens, "High-Level Synthesis: Current Status and Future Directions," *Proceedings of 25th Design Automation Conference*, pp. 477-482, June 1988.
3. F. D. Brewer and D. D. Gajski, "Knowledge-Based Control in Micro-Architecture Design," *Proceedings of the 24th Design Automation Conference*, pp. 203-209, July, 1987.
4. R. K. Bryton, R. Camposano, G. DeMicheli, R. Otten, and J. vanEijndhoven, "The Yorktown Silicon Compiler," in *Silicon Compilation*, pp. 204-311, Addison-Wesley, Reading, MA, 1988.
5. G. DeMicheli and D. C. Ku, "HERCULES: A System for High-Level Synthesis," *Proceedings of the 25th Design Automation Conference*, pp. 483-488, June, 1988.
6. S. Devadas and A. R. Newton, "Algorithms for Hardware Allocation in Data Path Synthesis," *Proceedings of IEEE International Conference on Computer-Aided Design*, pp. 768-781, July, 1989.
7. D. Freedman, *Markov Chains,* Springer-Verlag, New York, 1983.
8. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* Freeman, San Francisco, 1979.
9. Catherine H. Gebotys and Mohamed I. Elmasry, "Simultaneous Scheduling and Allocation for Cost Constrained Optimal Architectural Synthesis," *Proceedings of the 28th Design Automation Conference*, pp. 2-7, June, 1991.
10. C. H. Gebotys and M. I. Elmasry, "A VLSI Methodology with Testability Constraints," *Proceedings of 1987 Canadian Conference on VLSI*, Oct. 1987.
11. E. F. Girczyc, "Automatic Generation of Microsequenced Data Paths to Realize ADA Circuit Descriptions," PhD Thesis, Carlton University, July, 1984.
12. J. W. Greene and K. J. Supowit, "Simulated Annealing Without Rejected Moves," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. CAD-5, no. 1, pp. 658-663, January 1986.
13. L. J. Hafer and A. C. Parker, "Register-Transfer Level Digital Design Automation: The Allocation Process," *Proceedings of the 15th Design Automation conference*, pp. 213-219, June, 1978.
14. T. Hailperin, *Boole's Logic and Probability,* North-Holland, New York, 1986.
15. C. Y. Hitchcock and D. E. Thomas, "A Method of Automatic Data Path Synthesis," *Proceedings of the 20th Design Automation Conference*, pp. 484-489, June, 1983.
16. C-Y. Huang, Y-S. Chen, Y-L. Lin, and Y-C. Hsu, "Data Path Allocation Based on Bipartite Weighted Matching," *Proceedings of the 27th Design Automation conference*, pp. 499-504, June, 1990.
17. M. D. Huang, F. Romeo, and A. Sangiovanni-Vincentelli, "An Efficient General Cooling Schedule for Simulated Annealing," *Proceedings of IEEE International Conference on Computer-Aided Design*, pp. 381-384, November 1986.
18. C. Hwang, Y. Hsu, and Y. Lin, "Optimum and Heuristic Data Path Scheduling Under Resource Constraints," *Proceedings of the 27th Design Automation*

Conference, pp. 65-70, June, 1990.

19. R. Jain, A. C. Parker, and N. Park, "Module Selection for Pipelined Designs," *Proceedings of the 25th Design Automation Conference*, pp. 542-547, June 1988.
20. S. Karlin, *A First Course in Stochastic Processes,* Academic Press, 1973.
21. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671-680, May 1983.
22. R. Kling and P. Banerjee, "ESP: A New Standard Cell Placement Package using Simulated Evolution," *Proceedings of the 24th Design Automation conference*, pp. 60-66, June, 1987.
23. Ralph-Michael Kling and Prithviraj Banerjee, "Optimization by Simulated Evolution with Applications to Standard Cell Placement," *Proceedings of the 27th Design Automation Conference*, pp. 20-25, June, 1990.
24. G. J. Klir and T. A. Folger, *Fuzzy Sets, Uncertainty, and Information,* Prentice Hall, Englewood Cliffs, 1988.
25. T. J. Kowalski, in *An Artificial Intelligence Approach to VLSI Design*, Kluwer Academic Publishers, Boston, 1985.
26. D. Ku and G. De Micheli, "Relative Scheduling Under Timing Constraints," *Proceedings of the 27th Design Automation Conference*, pp. 59-64, June, 1990.
27. K. Kucukcakar and A. C. Parker, "MABAL: A Software Package for Module and Bus ALlocation," Technical Report CRI-88-61, University of Southern California, March 3, 1989.
28. F. J. Kurdahi and A. C. Parker, "REAL: A Program for REgister ALlocation," *Proceedings of the 24th Design Automation Conference*, pp. 210-215, June, 1987.
29. T. Lin, T. Hsu, and F. Tsai, "SILK: A Simulated Evolution Router," *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-8, no. 10, pp. 210-215, October 1989.
30. M. Lundy and A. Mees, "Convergence of the Annealing Algorithm," *Simulated Annealing Workshop*, Yorktown Heights, April 1984.
31. T. A. Ly and J. T. Mowchenko, "Applying Simulated Evolution to Scheduling in High Level Synthesis," *Proceedings of 33rd IEEE Midwest Symposium on Circuits and Systems*, August, 1990.
32. T. A. Ly and J. T. Mowchenko, "Applying Simulated Evolution to Data Path Allocation in High Level Synthesis," *Proceedings of Canadian Conference on VLSI*, pp. 6.4.1-6.4.8, October, 1990.
33. T. A. Ly, W. L. Elwood, and E. F. Girczyc, "A Generalized Interconnect Model for Data Path Synthesis," *Proceedings of the 27th Design Automation Conference*, June, 1990.
34. T. A. Ly and J. T. Mowchenko, "Bottom Up Synthesis based on Fuzzy Schedules," *Proceedings of the 28th Design Automation Conference*, June, 1991.
35. T. A. Ly and J. T. Mowchenko, "Comparing Simulated Evolution and Simulated Annealing using the Scheduling Problem in High Level Synthesis," *Proceedings of Canadian Conference on VLSI*, August 1991.
36. T. A. Ly and J. T. Mowchenko, "Applying Simulated Evolution to High Level Synthesis," *submitted to IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, June 7, 1991.
37. M. C. McFarland, A. C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems," *Proceedings of the IEEE*, pp. 301-318, February, 1990.

38. M. C. McFarland, "Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions," *Proceedings of the 23rd Design Automation Conference*, June, 1986.

39. D. Mitra, F. Romeo, and A. Sangiovanni-Vincentelli, "Convergence and Finite-Time Behavior of Simulated Annealing," *Proc. 1985 Cont. Dec. Conf.*, December 1985.

40. J. A. Nestor and G. Krishnamoorthy, "SALSA: A New Approach to Scheduling with Timing Constraints," *Proceedings of International Conference on Computer-Aided Design*, Nov. 1990.

41. J. A. Nestor, "Specification and Synthesis of Digital Systems with Interfaces," CMUCAD-87-10, Department of Electrical and Computer Engineering, Carnegie-Mellon, April, 1987.

42. B. M. Pangrle and D. D. Gajski, "Slicer: A State Synthesizer for Intelligent Silicon Compilation," *Proceedings of the IEEE International Conference on Computer Design*, October, 1987.

43. B. M. Pangrle, "Splicer: A Heuristic Approach to Connectivity Binding," *Proceedings of the 25th Design Automation Conference*, pp. 536-541, June, 1988.

44. C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization,* Prentice Hall, 1982.

45. N. Park and A. C. Parker, "SEHWA: A Program for Synthesis of Pipelines," *Proceedings of 23rd Design Automation Conference*, pp. 454-460, July 1986.

46. A. C. Parker, J. Pizarro, and M. Milnar, "MAHA: A Program for Data Path Synthesis," *Proceedings of the 23rd Design Automation Conference*, pp. 461-466, June, 1986.

47. P. G. Paulin, "High-Level Synthesis of Digital Circuits Using Global Scheduling and Binding Algorithms," PhD Thesis, Carlton University, January, 1988.

48. P. G. Paulin and J. P. Knight, "Scheduling and Binding Algorithms for High-Level Synthesis," *Proceedings of the 26th Design Automation Conference*, pp. 1-6, June, 1989.

49. Z. Peng, "Synthesis of VLSI Systems with the CAMAD Design Aid," *Proceedings of the 23rd Design Automation Conference*, pp. 278-284, June, 1985.

50. R. Potasman, J. Lis, A. Nicolau, and D. Gajski, "Percolation Based Synthesis," *Proceedings of the 27th Design Automation Conference*, pp. 444-449, June, 1990.

51. F. Romeo and A. Sangiovanni-Vincentelli, "Probabilistic Hill Climbing Algorithms: Properties and Applications," in *Chapel Hill Conference on VLSI*, pp. 393-417, 1985.

52. Y. Saab and V. Rao, "An Evolution-Based Approach to Partitioning ASIC Systems," *Proceedings of the 26th Design Automation Conference*, pp. 767-770, June, 1989.

53. A. Safir and B. Zavidovique, "Automatic Synthesis of Specific Image Processing Automata by a Simulated Annealing Based Design Space Search," *1989 Symposium on Circuits and Systems*, May, 1989.

54. J. Scheichenzuber, W. Grass, U. Lauther, and S. Marz, "Global Hardware Synthesis from Behavioral Dataflow Descriptions," *Proceedings of the 27th Design Automation Conference*, pp. 456-461, June, 1990.

55. C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf Placement and Routing Package," *IEEE Journal of Solid-State Circuits*, pp. 510 - 522, April, 1985.

56. H. Trickey, "Flamel: A High-Level Hardware Compiler," *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, pp. 259-269, March, 1987.

57. C. Tseng and D. P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems," *IEEE Transactions on CAD*, pp. 379-395, July, 1986.

58. R. A. Walker and D. E. Thomas, "A Model of Design Representation and Synthesis," *Proceedings of 22nd Design Automation Conference*, pp. 453-459, June, 1985.

59. R. A. Walker and D. E. Thomas, "Behavioral Transformation for Algorithmic Level IC Design," *IEEE Transactions on CAD of Circuits and Systems*, vol. 8, no. 10, pp. 1115-1127, October 1989.

60. G. Zimmermann, "MDS - The Mimola Design Methdod," *Journal of Digital Systems*, vol. 4, no. 3, pp. 337-369, 1980.